Graph layout using subgraph isomorphisms

Hofton, Antony Edward

Graph Layout using Subgraph Isomorphisms

Antony Edward Hofton

Ph. D. Thesis

October 2000

Research Institute in Software Evolution
Department of Computer Science
University of Durham

Abstract

Today, graphs are used for many things. In engineering, graphs are used to design circuits in very large scale integration. In computer science, graphs are used in the representation of the structure of software. They show information such as the flow of data through the program (known as the data flow graph [1]) or the information about the calling sequence of programs (known as the call graph [145]). These graphs consist of many classes of graphs and may occupy a large area and involve a large number of vertices and edges. The manual layout of graphs is a tedious and error prone task. Algorithms for graph layout exist but tend to only produce a 'good' layout when they are applied to specific classes of small graphs. In this thesis, research is presented into a new automatic graph layout technique. Within many graphs, common structures exist. These are structures that produce 'good' layouts that are instantly recognisable and, when combined, can be used to improve the layout of the graphs.

In this thesis common structures are given that are present in call graphs. A method of using subgraph isomorphism to detect these common structures is also presented. The method is known as the ANHOF method. This method is implemented in the ANHOF system, and is used to improve the layout of call graphs. The resulting layouts are an improvement over layouts from other algorithms because these common structures are evident and the number of edge crossings, clusters and aspect ratio are improved.

Acknowledgements

I would not have been able to produce this thesis without the help and support of a number of people. There may be others that have contributed by providing me with an interesting and enjoyable time throughout my research and for those not named below, I thank you all.

First of all I would like to thank my parents, Joan and Michael Hofton, for support both financially and emotionally throughout out my life. Without this help I would not be where I am today.

I would like to thank Malcolm Munro for being an excellent supervisor and supporting me when others may have not. Without this support I would not have made it this far into academia or my stay in Durham would not have been as enjoyable as it was.

My eternal gratitude must also go to Liz Burd, my father Michael Hofton, and my sister Michelle Hofton for proof reading this thesis and offering comments, constructive criticism and continual support.

I would like to thank all the staff and students at the department of computer science for making my life at university enjoyable. In particualar James Ingham for being an excellent housemate throughout my PhD years and my office mates Nicholas ('string player') Gold, Stephen ('windows') Rank, Phyo ('toast') Kyaw and Claire ('knight rider') Knight, all of whom provided many hours of interesting and amusing 'discussions'. Together with my friends Richard Hinds and Louise Williamson without whom my life would have been different.

TQ Environmental PLC in Wakefield sponsored this research. I owe all the directors and staff gratitude, without their trust and foresight to fund me; this research would never have been performed.

Finally, but by no means least, I would like to thank my examiners Keith Bennett and Mark Harman. Without their professionalism this thesis would never had been completed.

Table Of Contents

Table Of Figures

Figures

Tables

Algorithms

Copyright Notice

Declaration

No part of the material has previously been submitted for a higher degree in the University of Durham or in any other university. All the work presented here is the sole work of the author and no one else.

1. Graph Layout

1.1 Introduction

Laying out large graphs by hand is a difficult and laborious task. It is a task that should be automated; the theory is known as automatic graph layout. This thesis addresses the problem of automatic graph layout for graphs used in software engineering. These are used in the practice of providing a better understanding of programs by maintainers who are changing the code. When laying out graphs by hand there are various techniques that have to be modelled for a layout algorithm to be successful. This thesis concerns the application of these techniques to the automatic layout of software engineering graphs. These graphs, like many other types of graph, can quickly become unreadable using automatic layout algorithms.

Figure 1 shows a typical small software engineering graph, known as a call graph. This has many problems, the main ones being the high number of edge crossings and the overlapping of vertices. These problems are typical of the layouts produced from automatic layout algorithms and are collectively known as the 'Graph Layout Problem'.

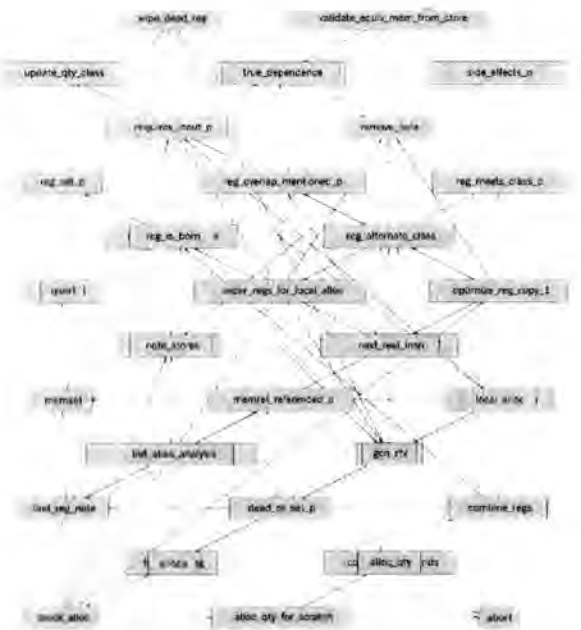


Figure 1 - A typical call graph layout

Applying standard layout algorithms, such as Sugiyama, Tagawa and Toda. [159], improves this. When Sugiyama is applied to Figure 1, the graph shown in Figure 2 is produced. However the edge crossings are still high in this diagram. In the following chapter it is suggested that maintainers look for common structures in graphs to aid them in understanding the software under consideration. These common structures can be used to improve the layout of the graph.

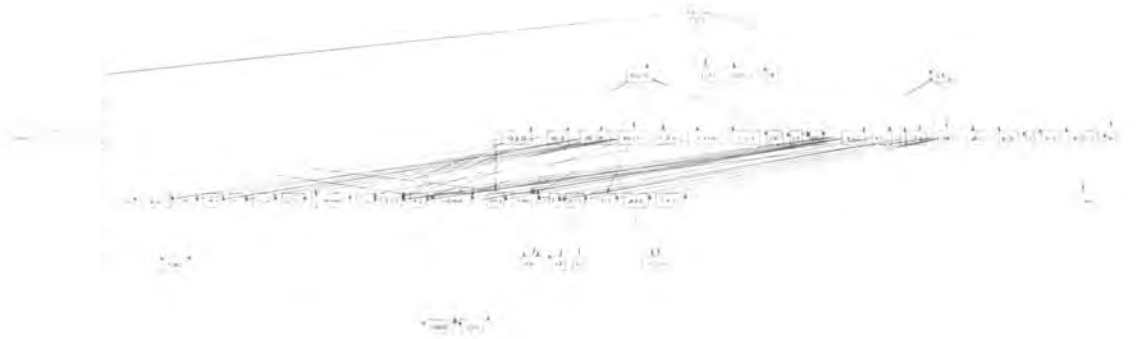


Figure 2 - A typical call graph laid out using the layout algorithm of Sugiyama et al.

In Chapter 4 of this thesis it is shown that when these common structures are given a standard layout, collectively known as common model graphs, they can be used to improve the layout of graphs by reducing the edge crossings and aiding understanding by making these structures become apparent. These techniques when applied to the graph in Figure 1 yield the much improved layout of Figure 3. It is a much improved layout because the common structures are clearly identifiable by the different coloured vertices. In addition, a reduced number of edge crossings and related vertices are placed together.

Figure 3 - A typical call graph laid out using the ANHOF system

This chapter puts graph layout into context in terms of software engineering, maintenance, reverse engineering and visualization. The problem is stated and the criteria for success are given. Finally the thesis structure is presented.

1.2 Software Engineering

The term software engineering was introduced at a NATO conference in the 1968 to discuss the 'software crisis' [121]. This 'crisis' was caused by machines becoming powerful enough to make the automation of everyday tasks feasible, thus causing larger applications to be built. Existing development techniques were not sufficiently robust because techniques for small systems development could not be scaled up. Projects ran late, over budget and were difficult to maintain. Whilst hardware was becoming cheaper, software was becoming more expensive. New techniques were needed to control the complex task of developing software.

Today this software crisis is not completely resolved. There are improvements in software engineering methods and techniques, development tools and in the skills of the I.T. staff. However, the demand for software outstrips the improvements in software productivity, and mistakes that were made in the 1960's are still being made today. Somerville [154] suggests some factors of a well-engineered piece of software are: -

- the software should be easily maintainable. Software is subject to regular change and be written and documented to aid this task,

- the software should be dependable. This means that it should perform as expected by users and not fail more than the specification suggests it will,

- the software should be efficient and not be wasteful on resources, and

- the software should offer an appropriate user interface.

Research into the development process is continuing. One of the first models produced was that of a waterfall by Royce [144]. Developers welcome it because the process is

visible and manageable. There are five phases to the model, each one leading to a change in the ones preceding it and is therefore an ongoing process. Royce [144] describes these processes as: -

- **requirements analysis and definition** - The system's functionalities, constraints and goals are established by consultation with the system users. Both users and the development staff define them in a manner that is understandable.

- **system and software design** - The system design process partitions the requirements of both hardware or software systems and also establishes an overall system architecture. Software design involves representing the software system functions so that they are transformed into one or more executable programs.

- **implementation and unit testing** - During this stage, the software design is realised as a set of programs. Unit testing involves verifying that each program meets its specification.

- **integration and system testing** - The individual programs are integrated and tested as a complete system to ensure that the software requirements are met. After testing, the software system is delivered to the customer.

- **operation and maintenance** - Maintenance involves correcting errors not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

This model is a general one and many processes are vague. Other models have been suggested, for instance the spiral model by Boehm [17], and the rapid prototyping model by Fairley [55]. An advantage of the waterfall model is that analysis and planning are performed before any major decision is made.

Whichever model is used graphs are used in many stages of the process. They are used as planning, design and maintenance tools. The layout of these graphs is crucial to the software engineering process.

1.3 Software Maintenance

After the software has been delivered it is inevitable that the software will need to be altered to implement changes to the specification or to correct errors. The term software maintenance describes this process. It is defined as, "*the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adopt the product to a changed environment*" [1]. These modifications range in size, from rewrites to accommodating new requirements to correcting coding errors.

It is perceived that it is impossible to produce a program of any size that does not need to be maintained [102]. With this in mind, programs should be designed to minimize problems with maintenance. Lientz and Swanson [105] suggest that large organisations devote at least 50 percent of the total programming effort to maintaining existing systems. In accordance with the above definition there are four types of software maintenance. These are: -

- perfective maintenance,
- adaptative maintenance,
- corrective maintenance, and
- preventative maintenance.

Perfective maintenance involves implementing new functionality or non-functional system requirements. This accounts for approximately 60 percent of all software maintenance [105]. An example of this is to change a menu order to cause it to be more user friendly.

Adaptive maintenance is maintenance that is required because of changes in the environment of the program. This accounts for approximately 18 percent of all software maintenance [105]. An example of such maintenance is where a system is changed to work on platforms different to the original specification.

Corrective maintenance is the correction of previously undiscovered system errors, and accounts for a further 17 percent [105]. An example of such maintenance is where a system is changed to correct an error discovered in its code.

Preventative maintenance is the updating of software to overcome future problems and to increase maintainability accounting for approximately five percent of all software maintenance [105]. An example of this the rewriting of a module because it is the source of many bugs.

Often errors present in the code are relatively easy and inexpensive to correct. Design errors are more expensive, mainly because they could involve re-coding several program components. If a customer changes his requirements, it can often lead to a redesign followed by rewriting large sections of the program, and is therefore expensive to correct.

Maintenance programmers have to cope with large and ever increasing volumes of software. Often the documentation for this software is not suited to the reader's requirements and may even be lost or incomplete. Here techniques of program comprehension and reverse engineering are used, both of which are discussed below. Graphs are used as a comprehension aid because they are an effective representation of the structure of the program. The layout of them is therefore crucial to the comprehension of the program.

1.4 Program Comprehension

Program comprehension is a vital part of the process of software maintenance. If a program is to be modified then one of the first steps required is to know what the program does. Studies have shown that 50 – 90 percent of maintenance time is devoted to program comprehension [156]. If documentation is present, then three and a half times as much time is put into studying the code than studying the documentation [60]. When programmers study code they are trying to, *"understand the intent and style of the programmer"* [60].

There are some applications in use today that were developed in the 1960's or 1970's, before the advent of development methods and before the word 'engineering' had ever been applied to programming. Often such systems have inadequate or non-existent documentation. When software is behind schedule, updating the documentation is not high on the developer's priority list, and it is difficult for later programmers to obtain knowledge of the system.

In cases like this program comprehension techniques are required. Techniques, such as viewing the program code or extra information generated from that code allow the programmer to gain knowledge about the program. There is often no alternative to reading the code. Understanding the code by reading is a function of the program size and its complexity [143]. Despite this, the method is effective. The sheer volume of information in the code often makes extracting the required knowledge a difficult task to perform. Clarity of the code is often helped by using indentation [150], using meaningful variable names [116], proper use of comments [178] and modularity of code [99].

According to Littman, Pinto, Letovsky and Soloway [107] there are two strategies that programmers apply in order to understand a program. The first strategy is to try understanding the whole program, known as "*the systematic approach*". The other is the "*as needed strategy*", where the programmer studies the part of the program as and when needed. There is no consensus of which method is best, but the authors conclude that a systematic approach is the best. However the original study involves small programs only. On a large program this may not be feasible. The approach used influences the knowledge the programmer achieves about the code. It is this knowledge that decides whether a modification is successfully achieved. The "*systematic approach*" acquires knowledge on the casual interactions of the program's functional components and the knowledge of the program structure (static knowledge) leading to a successful modification. However this static knowledge is not gained using the "*as needed approach*". Both of these strategies allow programmers to build up the information about a program in a mental model. There are strong and weak mental models [107]. Weak models only contain static knowledge and are therefore gained from the "*as needed approach*". However, strong mental models contain casual and

static knowledge about the program. These models can take many forms including chunks, hypotheses, and beacons.

Text structures are formed from the program's source code and its structure. It is dependent on its presentation, so it is improved by, amongst other ideas, indentation, and use of comments.

Chunks were originally described by Schneiderman [150]. He described a process where maintainers abstract portions of the source code in to chunks. These are then collected together into higher-level chunks. Chunks can be immediately understood or returned to later for revision.

Hypotheses were introduced by Brooks [25]. Hypotheses are a set of theories on what the program does and how it works. These are then rejected or refined until the correct set of hypotheses are found.

Beacons were introduced by Brooks [25] and further explored by Wiedenbeck [172]. These are recognisable or familiar features within the source code or other forms of knowledge. Beacons act as cues to the presence of certain structures or features and can be used in the forming or verification of hypothesis.

1.5 Reverse Engineering

Software documentation aids program comprehension and software maintenance. The documentation should be produced in accordance with the source code, and as part of the development process. Both should be passed onto the maintenance team. This is rarely the case, and often the documentation is not of any use to the maintenance team, due partly to its outdated nature. There are basically two types of documentation, development documentation and user documentation. Neither is produced with the maintainer in mind. Therefore the maintenance programmer reconstructs the useful documentation from the source code, in order to have adequate understanding of the system. This process is commonly known as reverse engineering, and is an example of the overlap between software maintenance and software engineering.

The term reverse engineering has its origins in the analysis of hardware, based on the practice of extracting the designs from a finished product [33]. Rekoff [142] defines reverse engineering as, "*the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system. These specifications are being prepared by persons other than the original designers, without the benefit of any of the original drawings ... for the purpose of making a clone of the original hardware system.*" Chikofsky and Cross [33] adapt this definition to apply to software engineering and is: -

"*Reverse engineering is the process of analysing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.*"

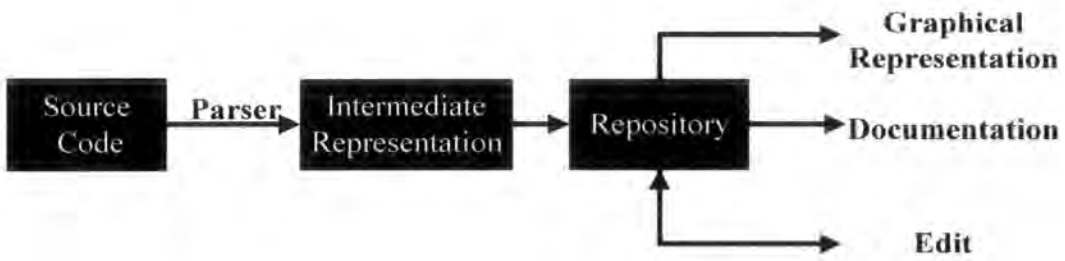


Figure 4 - A typical reverse engineering tool (adapted from [33])

Figure 4 above shows a typical reverse engineering tool. The tool takes a program's source code, and parses it. This generates an intermediate representation of the program that is then placed in a repository. An example form of the representation is a Prolog fact base. Various tools can be developed that allow the facts that are in the repository to be taken out and used to produce items such as new documentation.

Visualization techniques can be used to present the facts in the repository in a graphical manner. The justification for visualization will be presented below along with some of these graphical representations.

1.6 Visualization

The word visualize can be defined by the phase *"to make visual to the eye and mind"* [61]. This can be generalised and applied to software engineering as a process that aids the programmer in the understanding of a computer program. Program visualization aids in the process of understanding the complete program, not only the part visible to the user on the screen but its concept, its aims, and its structure.

Reading text is a special case of visual processing. However this is translated or interpreted as a character, word or a phrase level. Unfortunately such detail is too great and the brain will abstract each character, word or sentence into internal meaning or representation. Two methods of describing this process are beacons [25] and chunks [150]. Haber [75] in the late 1960's and early 1970's demonstrated the brain's great capacity for analysing pattern, colour and dimension. Software visualizations make use of this fact by presenting these program comprehension abstractions in a visual form. In studies by Cunnliff and Taylor [38] it was found that graphically presented code is faster and more accurate to comprehend. This was shown through experimentation by comparing a subjects understanding of programs represented both graphically and textually.

Effectively presenting large amounts of information in any form is challenging. Often there is restricted space to present this information. Where a computer is to be used, this space is restricted to the size of the screen. It is often possible to fill this space with so much information and detail that it completely overwhelms the user. It has been said that it is not the amount of information but how it is presented. The process of presenting this information is known as 'visualization'.

A definition of visualization is provided by Knight [97] and is *"visualization is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the phenomena under consideration"*. Myers [120] attempts to summarise the benefits of visualization as, *"The human visual system and human visual information processing are clearly optimised for multi-dimensional data. Computer Programs, however are conventionally presented in a one*

dimensional textual form, not utilising the full power of the brain". However further justification can be provided from research undertaken under the title 'information visualization'. These are summarised by Knight [97] : -

- being able to summarise a large amount of information in one view and thus providing an overview,
- being able to see correlations or patterns that may have otherwise been missed had only the figures or categorical data been used,
- trying to display structural relationships and context that may be more difficult to detect by individual retrieval requests, and
- providing an effective way of going between overview abstractions and the detail of the data.

There are many taxonomies that attempt to classify the many types of visualizations. ([120] and [133]). Myers suggests that program visualization can be classified into one of the following areas. These are particularly relevant to two-dimensional visualizations: -

- static code visualizations,
- dynamic code visualizations,
- static data visualizations,
- dynamic data visualizations,
- static algorithm visualizations, and
- dynamic algorithm visualizations.

One technique of software visualization and also a technique of static analysis of programs is the extraction of various graphs representing the program. Examples of graphs used in software engineering are flowcharts, control flow graphs, call graphs, and data flow graphs. A brief introduction to graphs, and in particular automatic graph layout, is given below.

1.7 Automatic Graph Layout

Graphs are used in many areas ranging from chemical structure diagrams to flowcharts in computer science. The field of automatic graph layout is diverse and fundamentally it can be divided into two areas; the description of the structure of the graph, collectively known as graph theory and, the methods of presenting them, collectively know as graph drawing.

Graph theory covers the terminology of graphs, in terms of the mathematics behind them, and their structure. It makes use of standard mathematical terminology in such areas as Cartesian co-ordinates and set theory. There are many general books and papers devoted to the area, e.g. [176] and [131].

Graph Drawing describes the process of formatting the graph so that is can be displayed to a user. It can be displayed on such media as paper or a visual display unit. In order to display the graph it has to be formatted from a raw list of vertices and edges to a defined layout that can be displayed. The defined layout is obtained using an automatic graph layout algorithm. When automatic graph layout became feasible new layout algorithms were devised. These take a graph of a certain class and lay out the vertices and edges of that graph taking various aesthetics into account. DiBattista et al. [42] provides an annotated list of references on the topic.

In the last couple of years there has been a consolidation of the graph drawing area. Many texts have tried to draw together the information and algorithms. Presenting the best and worse algorithms for each class of graphs. Research has started to be performed in where to apply each algorithm. Consequently there are growing numbers of books and papers that provide a good general discussion on the topic, such as [161] and [43].

In Chapters 2 and 3 the whole field of automatic graph layout will be presented. It will present the graph theory necessary for this thesis and, the various different classes of graphs. It will discuss the various automatic graph layout algorithms and many other areas of the automatic graph layout.

1.8 Problem Definition

In the domain of software engineering, graphs are used for displaying the structure of a computer program. For instance to display the information of which procedures call which (known as a call graph) or the flow of information through the program (known as a flowchart).

Earlier in this chapter it was shown that there is a problem in laying out graphs. A call graph was shown as an example (Figure 1, Figure 2 and Figure 3). This graph and other graphs used in software engineering do not fit into any single class of graph. They consist of many classes of graphs. The problem is not obtaining a layout of the graph; it is controlling the layout of the graph so that they are more comprehendible. There is also a lack of techniques that combine some of the solutions in the literature in one tool. It is necessary to develop an algorithm for laying out graphs in the domain of software engineering that can be used to draw many of the types of graph in that domain and can be customised to the users needs.

Many automatic graph layout algorithms work well with specific graph classes, e.g. tree, but do not scale up very well to larger graphs. Software engineering graphs are often made up of several of these graph classes. Current research in automatic graph layout cannot proceed unless layout algorithms concentrate on a specific graph type that is designed for a specific purpose and should stop concentrating on the general problem. It is therefore the intention of this research to improve automatic graph layout for a specific type of graph that is commonly used in program comprehension. If programmers are aiming to understand the whole program, a study by Jeffries [85] suggests that when comprehending the program that programmers have to read the program code, to do this they use a common method. They read it in the order in which it would be executed, main procedure first, then procedures called by the main procedure, and then procedures called by those procedures, etc. This knowledge of the structure is represented by a call graph and represents a top down approach to program comprehension. The graph type that will be the focus of this research will therefore be that of the call graph.

Schneiderman [150] suggested that maintainers 'chunk' together sections of code. They also look for key features in code known as beacons. Both these techniques can be applied to call graphs. When comprehending call graphs maintainers may look for common structures. Any layout algorithm for call graphs should therefore make these common structures appear in the final layout of the graph. These common structures also have other advantages. These can lead to the simplification of call graphs or can aid in the layout of them. These common structures have associated with them a 'good' layout. The original graph is broken up into these structures and others, called subgraphs. Each subgraph is then laid out using their associated layout algorithm, and the original call graph is rebuilt of 'well' laid out graphs and a 'well' laid out call graph is what remains. This is a form of the 'divide and conquer' method of automatic graph layout suggested by Messinger, Rowe and Henry [113].

It is obvious through reading several comparisons of automatic graph layout algorithms that the success of using these algorithms is measurable using various aesthetic properties. These properties are discussed later in this thesis. Thus the term 'well laid out graph' can be described in terms of its aesthetic qualities. Standard layout algorithms set standards that should be used when laying out graphs. Any new layout method should improve these standards.

1.9 Criteria For Success

This research in this thesis will produce an automatic graph layout algorithm/system that will: -

- identify the common structures in call graphs,
- produce well laid out call graphs that are to a high quality in terms of the metrics of the graph,
- be able to improve the layout of large call graphs,
- have the ability to describe the graph in a simple language,
- be able to detect various common structures that have been found to be present in many call graphs, and
- develop a prototype tool to show proof of concept.

1.10 Thesis Structure

In the next two chapters an overview of the literature in the areas of automatic graph layout algorithms, software engineering domain graphs and graph theory problems is given. These will be discussed under the following titles: -

Chapter 2 – Some Graph Theory
Chapter 3 – Some Graph Uses

In Chapters 4 and 5 the solution to the above problem is outlined. In Chapter 4 some of the common models present in call graphs are given, and a theoretical method of detecting them and using them to improve the layout of call graphs is outlined. This method is discussed under the title of 'the ANHOF method'. In Chapter 5 a description of an implementation of the method known as the ANHOF system is given. The titles of the chapters are as follows: -

Chapter 4- The ANHOF Method of Call Graph Layout
Chapter 5- Implementing the ANHOF Method

In Chapters 6, 7, and 8 the results of applying the ANHOF system are given. In Chapter 6 the ANHOF system is optimised by investigating the settings necessary to obtain its best performance. Whilst in Chapter 7 the system is compared with other layout tools, and its performance is discussed. In Chapter 8 the performance of the ANHOF system is given in terms of the metrics of the graphs it produces. These are compared with other algorithms. The titles of the chapters are: -

Chapter 6 – Tuning the ANHOF system
Chapter 7 – The ANHOF system at work
Chapter 8 – The performance of the ANHOF system

Chapter 9 draws conclusions from and summarises this research. It suggests further work and possible future expansions of the research. It is titled as follows: -

Chapter 9 - Conclusions and Further work

1.11 Summary

This chapter places graph layout in the context of software engineering, maintenance, reverse engineering, program comprehension and software visualization. It has defined the problem of automatic layout of graphs of a particular type used in the domain of software engineering, known as call graphs. This is followed by the criteria of a successful research project into the problem. Finally the structure of the rest of this thesis is given. In the next chapter the diverse area of automatic graph layout is presented. It will show that layout algorithms only work on a specific class of graph.

2. Some Graph Theory

2.1 Introduction

Many areas of science and engineering use graphs to represent systems comprising of a large number of interacting components. Chemists use graphs to model interactions between particles and engineers use them to represent very large scale integration (VLSI) circuits. In computer science they are commonly used to represent databases, semantic networks, knowledge representations and, in large programs, model control flow and module dependency. To generalise, graphs are used to model systems where the number of components is large but the components themselves are simple. They are described by Tamassia, DiBattista and Batini [162] as "*an effective documentation means and representation for both the designer and the user, they are a common language to express the requirements of the application in a formal way*".

In Chapter 1 a problem of laying out software engineering graphs was given. It suggested that layout algorithms work on specific classes of graphs. In the chapter below a summary of the underlying graph theory is given.

This thesis describes a process applying certain techniques which layout graphs automatically and an improved method of layout for domain specific graphs, the domain being restricted to software engineering. This chapter of the thesis describes current research in the field of automatic graph layout, which is a diverse field. In particular this survey looks at graph theory, graph specification languages, automatic layout algorithms, graph tools, graph metrics, graph aesthetics, the different graph types and graph isomorphism. The fields are shown in Figure 5.

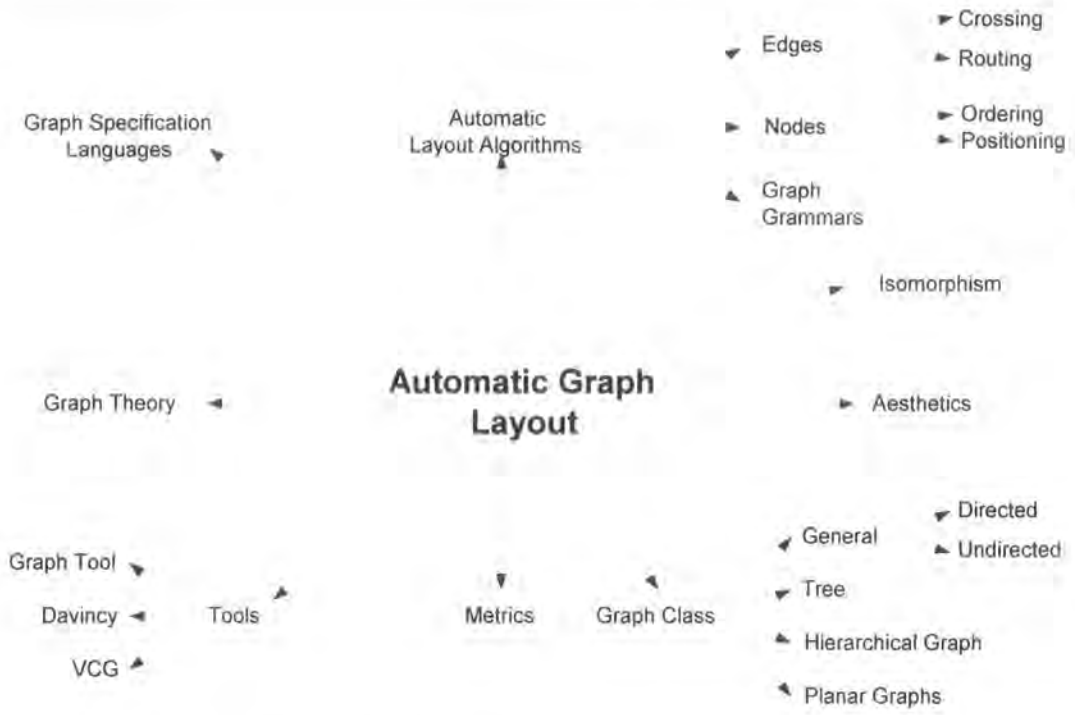


Figure 5 - The area of automatic graph layout

Graph layout can be performed manually but is laborious and difficult and is therefore ideal for automation using computers. However this causes many problems. For instance what is a 'good layout' and what constitutes a 'good graph layout'? In this chapter a summary of the literature in the areas of graph theory, layout algorithms, graph types, metrics and aesthetics is presented. In the next chapter the graph theory given below is used to discuss various types of software engineering graphs, graph tools and graph specification languages. In addition, graph isomorphism is discussed in Chapter 3.

2.1.1 Definitions

Throughout this thesis the term 'graph' is used. It is a term that underpins all the graph theory that is used to design and describe graph layout algorithms. Before proceeding further it is necessary to define a general graph. There are many books on the general theory of graphs, two being [176] and [73]. They both agree on this definition of a graph: -

"A Graph G = (V, E) is a mathematical structure consisting of two sets V and E. The elements of V are called vertices (singular vertex) or nodes, and the elements of E are called edges; each edge has a set of one or two vertices associated to it, which are called its end points"

In relation to this the following terms from graph theory will be used in this thesis: -

- **path** – a method of getting from one vertex to another. It is sequence of edges in which no vertex appears more than once.
- **strongly connected graph** - if for all x, y ε V, there is a path from x to y and a path from y to x.
- **complete graph** – a graph in which every two distinct vertices are joined by an edge.
- **labelled graph** – a graph in which the vertices have been assigned an identifier either by a function or manually.

Another definition that is more formal and commonly quoted is: -

A graph G is a tuple (V, E) where V is the set of elements called vertices and E is the set of elements called edges and $E \subseteq V \times V$

As stated above this thesis is concerned with the automation of the layout of graphs used in the domain of software engineering. Graph layout is concerned with the positioning of the elements of the graph, the vertices and the edges. The vertices are given a position on a Cartesian plane. The edges are given a set of coordinates using the same Cartesian plane in which to pass through in order to fulfil the goals of the layout. The route may have to obey various criteria, for instance not to cross other edges or vertices. An automatic graph layout algorithm describes the formal process of laying out a graph.

2.2 Automatic Graph Layout Algorithms

According to Bertolazzi, DiBattista and Liotta [12] there are two approaches to automatic graph layout algorithms, the declarative approach and the algorithmic approach. The algorithmic approach consists of designing special purpose layout algorithms; each algorithm is devoted to solving the layout problem for specific sets of requirements and specific graph structures. The other is the declarative approach consisting of devising languages for describing sets of requirements, and of using logic programming to construct diagrams that fit the given requirements. Many of the automatic graph layout algorithms for large general graphs use a combination of the declarative and algorithmic approaches, first suggested by Lin and Eades [106].

There are many graph layout algorithms. DiBattista, Eades, Tamassia and Tollis [42] provide a summary of some of the algorithms that have been published. It has been the basis of much of this research and of others. According to Tamassia et al. [162] layout algorithms can be categorised by: -

- the class of graph they apply to,
- the graphic standard used to layout the graph,
- aesthetics,
- constraints, and
- computation complexity.

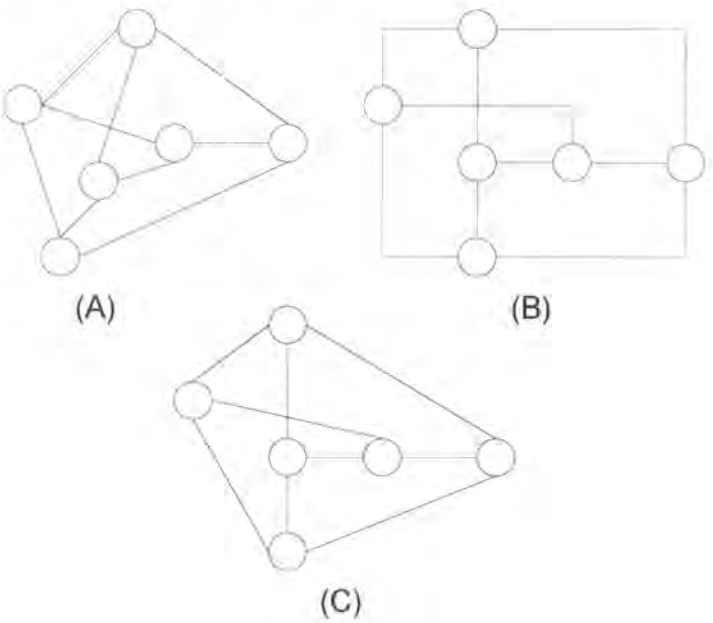
Generally graphs can be categorised into four classes. Future sections will take each class in turn and discuss their layout. The classes are: -

- general undirected / directed,
- trees,
- planar, and
- hierarchic.

When graphs are laid out either manually or by machine the visual representation that is produced is called a diagram. Three standard types of diagrams are produced each with their associated technique for producing them. The standards are: -

- **straight line standard** – laying out the vertices, and edges as straight-line segments, e.g. trees.
- **grid standard** – place vertices on grid points and the edges follow the grid.
- **mixed standard** – place vertices on grid points and route edges as straight-line segments.

Figure 6 shows a six-vertex, nine-edge graph laid out using the various standard methods for creating a diagram of the graph.



(A) Straight line standard, (B) Grid Standard (C) Mixed Standard

Figure 6 – Layouts of the same graph using the various layout standards

2.2.1 A General Graph Layout Algorithm

When studying the literature it becomes obvious that the algorithms presented have common steps. The graph layout algorithms found in the literature can be summarised by the following general algorithm. The rest of the chapter will present more specific

algorithms. The general algorithm presented below is meant as a general guide to the literature and to developing a 'successful' algorithm. Please note that step three does not always apply: -

1. Order graph vertices - Rank or sort them into an order that is based on their connectivity.
2. Position vertices using the order.
3. Reposition vertices with the aesthetics in mind.
4. Route and draw edges.
5. Display graph.

There are many methods of performing steps 1 to 3 depending upon the class of graph. They are discussed in later sections dealing with the class of graphs.

2.2.2 Edge Routing

Edge routing (step 4) is an important problem in graph layout. Many studies have been performed in engineering in the design of circuit boards (VLSI) but little research has been done in relation to graph theory and automatic layout. Many other areas have been studied and shown to have similar problems, e.g. a robot finding its way through a maze, and many algorithms are based on this research.

Dobkin, Gausner, Kotsofios and North [47] provide an introduction to the problem of edge routing, and also provides a definition. The definition is: -

"*In a given polygon P containing a set of holes corresponding to the obstacles S, given two points p and q (inside P) find a path L from p to q that stays within P avoiding all obstacles in S. Map L onto a plane in P using geometric structures.*"

Generally edges are added in a way that clearly exhibits vertices without adding clutter or deceptive artefacts. Therefore a route for the edge must be found. Hsu [82] suggested that a route is either routed on a topological plane or is mapped on to a geometric plane, e.g. a grid. According to Dobkin et al. [47] a good route should: -

- avoid other vertices in graph,
- stay close to shortest path,
- not turn sharply, and
- avoid any unnecessary inflections.

A general solution to the problem that could be applied to graph layout, is provided in the following algorithm: -

1. Find the shortest path in polygon P from p to q such that all obstacles are avoided.
2. Fit path onto plane.

There are many solutions to find the shortest path from p to q; a few are given in [47]. Edges should bend to avoid touching incident vertices, Dobkin et al. suggest that edges can be drawn using polylines or curves such as bezier splines to aid this.

2.3 Graph Classes

All graphs have the property of being either directed or undirected. Directed graphs are used commonly in software engineering to represent a notion of information flow. Later chapters show that this thesis concentrates on improving the automatic layout of directed graphs. Undirected graphs pose many problems. An edge between two vertices can be traversed in either direction. This lack of flow means that they are not easily traversed automatically; there is no natural order to the vertices. The general algorithm presented above cannot be used to design algorithms for them.

Directed and undirected graphs are largely general graph types. They are often too general to apply automatic layout algorithms to. It is better to restrict the definitions of the graph types further. The layout algorithms produce better results if the exact structure of the graph is known. Therefore, automatic layout algorithms tend to apply to hierarchic, tree, and planar graphs. The section below provides a definition of the general classes of directed and undirected graphs, and discusses the other classes of graphs, giving automatic layout algorithms and associated problems for all the classes.

2.3.1 General Classes

In the section below the general classes of undirected and directed graphs are defined and automatic layout algorithms are given.

2.3.1.1 Undirected Graphs

2.3.1.1.1 Definition

Undirected graphs are commonly known as general graphs. This definition is always quoted as a starting point for definitions of other types of graph. A good definition is provided by Polimeni and Straight [131] and is: -

"An undirected graph consists of a finite non-empty set V and a set E of two element subsets of V. The Set V is called the vertex set of Graph G, while E is called the edge set of G. The graph is denoted by ordered pair G=(V, E)"

This definition is another method of expressing the formal definition of a graph given above. This is because when a graph is referred to without further definition, it is generally a graph without any flow to the edges that is being referred to.

An example of an undirected graph is given in Figure 7.

V={u,v,x,y}
E={uv, vx, xy,yu,ux}

Figure 7 - An undirected graph

2.3.1.1.2 Automatic Layout Algorithms

Until 1984 there was no clear direction for laying out general undirected graphs, as there were very few ideas on what to do with them. This was because there was very little information to give the vertices an order and consequently very difficult to layout. When printed circuits were just beginning to be made smaller by VLSI, graphs were used to help lay out such structures and automatic techniques were being heavily researched. Quinn and Breuer [140] discussed the application of particle physics and springs to layout. They thought that they could apply weights (components) to springs (edges) before the system was allowed to reach a state of equilibrium yielding the ideal layout of a PCB. Eades [50] applied this technique to graph theory and this became known as the spring embedder model. Since then, there have been several variations on this theme; some are given below.

Eades's method uses an analogy to physics. Vertices are treated as mutually repulsive charges and edges as springs connecting and attracting the charges. Starting with an initial placement of vertices, the algorithm iterates the system in discrete time steps computing the charges between the vertices, updating their position accordingly. The algorithm stops after a fixed number of time steps. The problem with this idea is that there is little chance of a convergence of the algorithm and therefore terminating on its own. If the number of time steps is too small the quality of the layout is poor and if the number is to large, time is wasted. Kamada and Kawai [89] (KK) refined this algorithm, introducing an optimal edge length k. Vertices are updated by moving them one at a time. The advantage of this system is that it converges and therefore finishes automatically. However both of these methods have a problem in that the changes of state only affect local areas (local minima), and not the whole graph.

In statistical mechanics a system of randomness is introduced known as simulated mechanics. It differs from standard iterative improvement methods by allowing moves that spoil, rather than improve, the temporary solution. This improves the problem of local minima by using rules similar to those that define how liquids are cooled to a crystalline form. An arbitrary state is computed. Any downward move is accepted, while upward moves are accepted with a probability depending on a current temperature. Initially the system has the ability to perform arbitrary moves because the

temperature is still high. Later the probability of choosing a next state with more energy approaches zero as the temperature is lowered. This system is implemented in an algorithm by Davidson and Harel [40] (DH). It achieves aesthetically pleasing results on small and medium graphs. Frauchterman and Reingold [62] (FR) modify Eades's algorithm refining the forces on the springs by implementing a simple cooling schedule. Defining the distance a vertex can travel as being dependent on the current temperature.

The above may be regarded as the core papers on force directed placement techniques. In recent years there have been techniques that have tried to combine them, such as Frick, Ludwig and Mehldau [63] (GEM), combining the advantages of each.

Another method is to use a method suggested by Tunkelang [163] (TU), known as the incremental approach. Tunkelang uses a template of 16 locations. These are the eight local neighbour positions and eight positions at distance d. Tunkelang inserts the vertices one after another in some precompiled order, breadth first from the centre of the graph. For a new vertex Tunkelang checks the template positions of each of its neighbours and the corners of the screen as candidate positions and chooses the best. After each insertion, fine-tuning is applied. All neighbours of the current vertex are checked for an improved position.

A comparison of these algorithms can be found in Brandenburg, Himsolt and Rohrer [23]. Here the algorithms are tested obtaining the following information: -

1. run time,
2. the ratio of the length of the shortest and longest edges,
3. standard deviation of the edge length,
4. the number of edge crossings,
5. the distribution of vertices,
6. the ration of farthest and nearest pair of vertices, and
7. the area of the graph.

The following is a summary of the results: -

1. the algorithms are stable against random input graphs,
2. KK, FH, FR and GEM without crossing optimisation produce similar looking diagrams,
3. TU often gives different layouts to the others,
4. DH is the most flexible and also the most time consuming,
5. KK and GEM are the fastest,
6. FR is fast on small graphs, and
7. KK produces smooth layouts with a low ratio of the longest and shortest edges and a small derivation of the edge length.

Brandenburg et al. [23] concludes there is no clear winner. They suggest that the algorithms are applied in the following order: -

1. KK or GEM,
2. TU or FR, then
3. DH.

2.3.1.2 Directed Graphs

2.3.1.2.1 Definition

A directed graph is used in structures where flow needs to be represented, for example in control flow graphs [101] and call graphs [145]. A directed graph is a general term that covers a variety of graphs. Papers and texts tend to define each of these types separately and forget about a definition of directed graphs. However a good definition is provided by Polimeni and Straight [131] and is: -

"A directed graph (digraph) D consists of a finite nonempty set V, together with a subset E of the product set V × V. We call V the vertex set of D and E the edge set of D. The digraph D is donated by the ordered pair (V, E)."

This differs from the general definition of a graph given above. The edge set (E) is not a two-element subset of the product subset. Figure 8 gives an example of a directed graph.

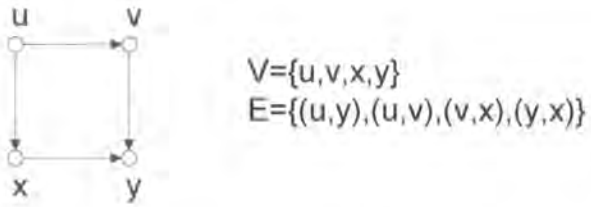


Figure 8 - A directed graph

2.3.1.2.2 Automatic Layout Algorithms

A method of laying out directed graphs is to use a hierarchical structure. One such method is presented by Gansner, Koutsofios, North and Vo [67]. He uses techniques more commonly applied to network flow and design to rank the vertices. This ranking generates a simple hierarchical representation of a directed graph. Their algorithm is simplified here: -

1. Rank vertices to obtain the level they are on.
2. Order vertices order the vertices on their vertices.
3. Position vertices
4. Make splines (Route Edges)

However, there are few algorithms for laying out general directed graphs because it is better to develop an automatic layout algorithm for a specific type of graph. There are attempts by Sugiyama and Misue [157] to apply the spring embedding algorithm (force directed approach) described above.

2.3.2 Trees

A tree can be either a directed or undirected graph. Tree data structures are commonly used in computer science usually in their directed form. Many texts cover the subject, a standard one being by Wirth [177]. They are commonly used to represent or store

hierarchical data. The flow of the graph represents the amount of data being stored, increasing or decreasing as levels are traversed. Many algorithms are available which manipulating them. A layout of a tree is often a good representation of the design of a computer program.

This section describes the current work on automatic tree layout. A tree is defined, the problems associated with tree layouts are given and an automatic layout algorithm is presented.

2.3.2.1 Definitions

There are many definitions of a tree. A mathematical definition given by Polimeni and Straight [131] is a "*Connected Acyclic Graph*". Whilst this is a good general definition, it does not represent the hierarchical nature of a tree. A better definition is provided by Aho, Hopcroft and Ullman. [4] and is: -

"A tree is a collection of elements called vertices, one of which is distinguished as a root, along with a relation ('parent hood') that places a hierarchical structure on the vertices.

A recursive definition is: -

1. *A single vertex by itself is a tree. This vertex is also the root of the tree.*
2. *Suppose n is a vertex and T_1, T_2,... T_k are trees with roots n_1, n_2,.., n_k respectively. We can construct a new tree by making n be the parent of vertices n_1, n_2, ..., n_k. In this tree n is the root and T_1, T_2, T_k are the sub trees of the root and T_1, T_2.... T_k are the sub trees of the root. Vertices n_1,n_2,..., n_k are called children of vertex n."*

A tree is therefore based on a hierarchical system, a system of levels. Other general definitions needed are the height of a tree and the width of the tree. The former is the number of levels in the tree. The width of a tree is the maximum number of vertices on any level within that tree.

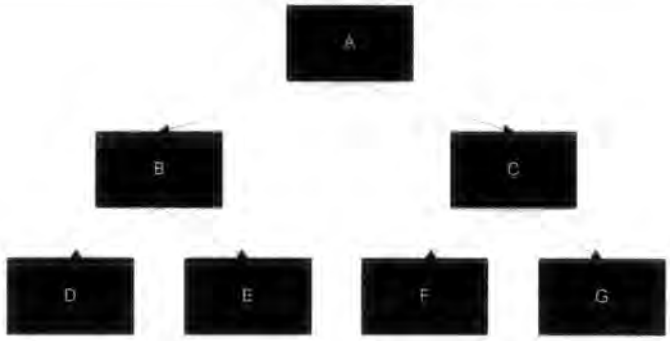


Figure 9 - A basic tree

In Figure 9 the height is three and the width is four. Vertex A is the 'Parent' of vertex B that is also the parent of vertices D and E etc. Vertex E cannot be connected to vertex A directly and still be a tree. Vertex A is on level one, vertices B and C are on level two and so on.

2.3.2.2 Classes of Trees

When studying the structure of several trees there are three distinct classes of tree. These are defined in [165] under the following titles: -

1. **dense tree** - this is a tree of height d and has a minimum number of vertices equal to 2d -1.
2. **degenerate tree** - this is a tree with one vertex per level.
3. **sparse tree** – a tree that has both degenerate and dense sections to it.

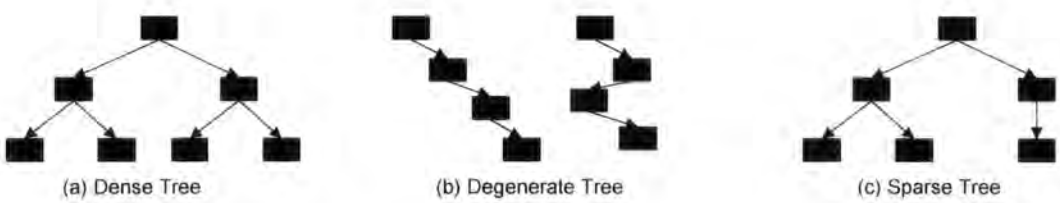


Figure 10 - The different classes of trees

2.3.2.3 Common Types of Tree

Table 1 shows some of the more common, but different types of tree.

Tree Type	Definition
Binary Tree	A finite set of elements (vertices), which either is empty or consists of a root (vertex) with two disjoint binary trees.
Multiway trees (n-ary tree)	A finite set of elements (vertices), which either is empty or consists of a root (vertex) with n or fewer of disjoint trees.
Balanced Tree	An optimisation of a tree which aims to keep equal numbers of items on each sub tree of each vertex so as to minimize the maximum path from the root to any leaf vertex. As items are inserted and deleted, the tree is restructured to keep the vertices balanced and the search paths uniform. They are mainly used in search trees, because they can make searching quicker when a search is confined to half the data. However, this is a strict definition of a balanced tree. Keeping a tree balanced is often time consuming and quite often removes the benefits of a balanced tree. A more frequently used definition is that of an AVL tree.
AVL tree	This tree is named after its inventors, Adelson-Velskii and Landis [3]. The definition is, "*A tree is balanced if and only if for every vertex the heights for its two sub trees differ by at most one.*"

Table 1 - The common types of tree

2.3.2.4 Problems Associated With Tree Layout

According to Vaucher [165] there are three main problems associated with tree layout: -

- **positioning** - this method relies on the computation of the X and Y co-ordinates of vertices. The X co-ordinates rely on the position of the neighbours, which are not easily obtained.
- **sequential printing** – the characteristics of standard printers require vertices to be printed sequentially in left to right or top to bottom order. This cannot be achieved with recursive algorithms because they don't work in a sequential manner.
- **overflow** - due to the limited width of a printed page the area required to print trees often exceeds the page area.

2.3.2.5 Aesthetics

A comprehensive summary of the aesthetics that should be applied when laying out trees is provided by Bloesch [14]. A brief summary is provided below: -

1. sibling vertices should have their top edges aligned horizontally,
2. sibling vertices should be laid out in the same left to right order as their logical order,
3. parent vertices should be centred over the centre of their leftmost and rightmost children,
4. a sub tree should be laid out in the same way no matter where it appears in a tree,
5. no edge joining the centre of the bottom with the centre of the top of a child should cross any other such edge or vertex,
6. all vertices that share a level should be separated horizontally by at least a distance $p>0$. Note: for the purposes of this aesthetic a vertex is considered to extend a distance $q>0$ above its edge, and
7. each vertex should be separated vertically from its parent by exactly a distance q. If vertices are composed of lines of text on a bit-mapped display, then q should be a multiple of the line height.

2.3.2.6 Automatic Layout Algorithms

There are many algorithms for the automatic layout of trees, two being [171] and [166]. A general algorithm is given by Bloesch [14]. In his paper he provides two algorithms that lay out trees in two different manners but arrive at a similar result. The general methods are as follows: -

- a post order traversal of a tree and for each level in the tree the computer stores the rightmost position at which a vertex has been placed, and positions a vertex either at some position defined by its parent's nominal position or the rightmost available position (following Vaucher's work) (Method 1), and
- a post order traversal of a tree, placing sub trees of a vertex so that they are some predefined distance x apart. Each branch has associated with it a left and right outline, once all sub trees have been placed, precede left to right positioning branches so that they are p units apart. (Method 2).

The above methods are for binary trees but can be easily applied to multiway trees. Bloesch [14] lists the advantages for both as the following: -

- method 1 is slightly faster,
- method 1 uses less storage space, and is therefore better for larger trees (more levels), and that
- method 2 has better aesthetics because the graph is often less wide and aesthetic four is always upheld . This is not the case in method 1.

In tests method 1 proved to be the easiest to implement and understand. The differences in aesthetics between the methods are not visible (see Figure 11). In tests the metrics generally are better for graphs produced using method 1. In conclusion method 1 is better.

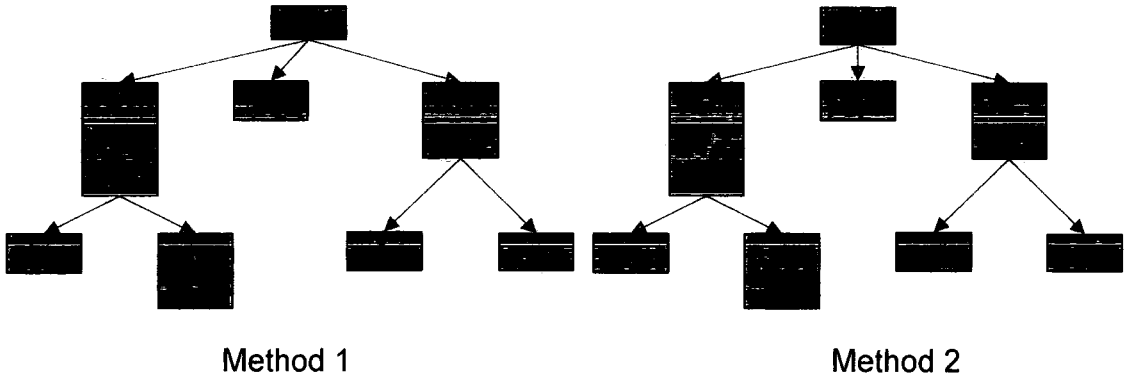


Figure 11 - A graph laid out using the methods above

2.3.3 Hierarchic Graphs

Hierarchies are common, for instance, many companies have hierarchic management structure. Wilson [175] provides a bibliography of hundreds of uses of hierarchies. In computer science they are used in PERT networks and call graphs. They all have common properties of being directed and acyclic.

This section provides a definition of a hierarchy and of a proper hierarchy. It discusses methods of representing and forming hierarchies as well as an algorithm for laying them out.

2.3.3.1 Definitions

Warfield [168] performed much of the early work on hierarchical graphs, including a widely accepted definition of a hierarchy. The definition has been slightly modified by various people. The one that is most often quoted and used is by Sugiyama et al. [159] and is: -

"*A directed graph (V, E) where V is called a set of vertices and E a set of edges which satisfies the following conditions.*

1). V is partitioned into n subsets, that is

$$V=V_1 \cup V_2 \cup \ldots \cup V_n$$

Where V_i is the i^{th} level and n the length of the hierarchy

2). Every edge $e = (v_i, v_j) \in E$ where $v_i \in V_i$ and $v_j \in V_j$ satisfies $i<j$ and each edge in E is unique

The notation used is G=(V, E, n). An n level hierarchy is called "proper" when it satisfies the following: -

3). E is partitioned into n-1 subsets that is

$$E=E_1 \cup E_2 \cup \ldots \cup E_{n-1} \ (E_i \cap E_j = \varnothing, I \neq j).$$

Where $E_i \subset V_i \times V_{i+1}$, $i=1,\ldots,n-1$

4). An order σI of Vi is given for each i, where the term "order" means a sequence of all vertices of V_i; $\sigma=v1v_2\ldots v_{|Vi|}$ ($|V_i|$ denotes the number of vertices of V_i). The n-level hierarchy is denoted by $G=(V,E,n,\sigma)$ where $\sigma=(\sigma_1,\ldots,\sigma_n)$."

This is different to the definition given by Warfield [168] in the following two points: -

1. in the definition by Sugiyama et al., edges are directed with ascending orders of levels, whist descending in Warfield's, and

2. in the definition by Sugiyama et al., orders of vertices are explicitly specified by σ. The orders are not specified in Warfield's.

With the exception of relations there is little difference between a tree and a hierarchical graph. In a hierarchy, each level will be related in some way, whereas the subtrees of the left and right sides of the root of the tree have little in common. A vertex on one level in a hierarchy can link to another up or down level(s). This is not possible in tree structure.

2.3.3.2 Methods of Forming Hierarchies

Warfield did much of the early work on hierarchies. He consequently published work on forming hierarchies in Warfield [167]. His method is a two-step process. The first step develops what he calls the subordination matrix showing all the inter-relations. From this it is possible to calculate which levels are subordinate to each other, and therefore form the hierarchy. This is therefore the second step.

Once the hierarchy is formed it is possible to form the interconnection matrix. From this several operations are possible for instance testing for planarity, obtaining the number of edge crossings, and laying out a hierarchy.

2.3.3.3 Automatic Layout Algorithms

There are algorithms for laying out hierarchical graphs; a few can be found in [42]. The most important ones are by Carpano [30] and Sugiyama et al. [159]. In a study of the tools available most have implemented the algorithm suggested by Sugiyama et al. However both researchers follow the general algorithm given in section 2.2.1. The differences lie in the methods used to solve steps 2 and 3. The steps are: -

1. form a proper hierarchy,
2. permute the orders of vertices in each level to reduce the number of edge crossings,
3. position the vertices horizontally, then
4. draw a two dimensional picture from these positions to the easily calculated level positions.

Sugiyama's presents both theoretical and heuristic approaches in developing algorithms for steps three and four. Details can be obtained from the paper [159]. The priority layout method is a heuristic to find the horizontal positions of the vertices (step 3). The approach works by reordering the vertices so that the most connected are at the start of a row. Details can again be found in the paper.

2.3.4 Planar Graphs

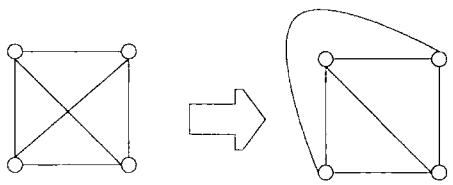
Planar graphs are commonly used in the areas of VLSI and circuit layout. This is because crossing of tracks in a circuit causes a short, and therefore is likely to make the circuit unusable. Soukup [155] provides a general introduction to circuit layout, and Bhatt and Leighton [13] provides a good introduction to the problems that arise in VLSI. Planarity is also a desirable property in graphs and diagrams as edges are easier to follow if they do not cross. This section provides a definition of planar graphs and a summary of the automatic layout algorithms available.

2.3.4.1 Definition

Polimeni and Straight [131] provide a general definition of a planar graph and is: -

"*If a graph is represented in the plane so that edges intersect only at incident vertices it is said to be planar.*"

An example of a planar graph is given in Figure 12.



(a) Normal graph (b) Planar representation

Figure 12 – A normal graph and its planar representation

2.3.4.2 Automatic Layout Algorithms

Automatic layout algorithms follow one of two theories: -

1. use PQ-trees to give the right ordering of the graph; or
2. try to embed the graph to a grid.

An algorithm that uses PQ-trees is presented by Chiba, Nishizeki, Abe and Ozawa [32], it is based on the vertex addition of Booth and Lueker [20]. It works in linear time, and is a two-phase process. The algorithm is complicated, and does not go as far as giving each vertex an x and y co-ordinate. It gives the vertices an order so that they can be embedded on a plane.

An algorithm that embeds a planar graph on a grid is presented by Tamassia [160]. It tries to embed the graph so that each edge has the minimum number of bends. It does this by applying minimal cost techniques to each edge. One possible problem is that the algorithm only works on four-planar graphs. These are graphs where each vertex has only four other edges coming or going from it.

2.4 Aesthetics

Diagrams are used to describe many things. Amongst other things they may be used as design or documentation tools. In each case they are used as a way of conveying information to a user in a clear and concise manner. One way in which to improve diagrams is to use an automatic layout algorithm. Batini, Furlani and Nardelli [8] suggest that they are commonly used for the following reasons: -

- reduction of production and maintenance costs,
- increase the expressive power of the diagrams,
- standardisation of the graphic project documentation, and to
- increase the communication between the designer and user.

Automatic graph layout algorithms produce a diagram, which represents an underlying graph structure. The aim of the algorithm is to organise the underlying graph structure in such a way that it is easier to read, understand and use. Designers of such algorithms use aesthetics to help the process, and claim that by doing so they help the user to understand and memorise the information contained in the graph. The following section presents a summary of the area of aesthetics. It defines an "aesthetic" and provides a summary of the aesthetics used in the literature.

2.4.1 Definition

There are few formal definitions of an aesthetic in the texts. The Oxford dictionary [61] provides a general definition of: -

"*A set of principles of good taste and the appreciation of beauty*"

This definition can be applied to graph layout. The only formal definition found is attributed to Coleman and Stott-Parker [35]: -

"*A measure of desirability in graph layout that is intended for human consumption.*"

2.4.2 Aesthetics in Graphs

A good visual representation of an area is related to the users' mental model. A mental model is the idea the person has of an external object or event. The goal of the representation is to give a correct idea of a system. Therefore the goal of any layout is to aid in giving a good mental map of an object or event. There are three important features of a diagram that help in creating this mental map [8], and these are: -

- readability
- relevance
- comprehensibility

Readability is the only evaluable feature of the three; the others depend on the area where they are being applied. Readability is evaluable by a deterministic approach, finding physical parameters related. A good diagram should give clear information about the associated object. This is known as the readability of a diagram. There are two types of readability: -

- conceptual reading – the structural properties of a graph.
- graphic readability – the layout of the diagram.

There are few studies of either area. Batini et al. [8] studied graphic readability in two-dimensional graphs, finding the first factor that influences readability is the set of rules and conventions that are used to layout the diagram. These are known as standards. Tamassia et al. [162] used this research to suggest that all diagrams belong to one of three standards. The standards are given below or are illustrated in Figure 6.

- straight line standard – where all the connections between symbols are straight lines.
- grid standard (orthogonality) – where connections run along the lines of a grid.
- mixed standard – a combination of the straight line and grid standards.

Once a graphic standard is established it is important that the criteria for achieving graphic readability (aesthetics) are detailed. They can be divided into two groups: -

- aesthetic features – concern the shape of the diagram, independently from the meaning of the symbols.
- semantic constraints – the layout rules for symbol placement, which allows the modelling of the unsupported semantic aspects, e.g. clustering of certain objects.

These standards and aesthetics can be categorised by the area of the graph that they affect. An aesthetic or constraint may be: -

- local (L) / global (G) – local when it refers only to a part of the diagram, global otherwise.
- hierarchical (H) / flat (F) – hierarchical when it concerns the relative position of a set of symbol, flat otherwise.

Batini et al. [8] summarise a study of two hundred diagrams against these categories the results of which are summarised in Table 2, Table 3 and Table 5. Table 4 shows the class of graph referred to by each aesthetic.

Acronym	Aesthetics	Category
Area	Minimisation of the area occupied by the layout.	G & F
Balan	Balance of the diagram with respect to the vertical axis or horizontal axis	G & H
Bends	Minimisation of the number of bends along the edges.	G & F
Convex	Minimisation of the number of faces drawn as a convex polygon	G & F
Cross	Minimisation of Crossings between edges	G & F
Degree	Vertices with high degree in the centre of the layout	L & F
Dim	Minimisation of differences among vertices dimensions	G & F
Length	Minimisation of global length of edges	G & F
MaxCon	Minimisation of the length of the longest edge	G & F
Symm	Symmetry of sons in hierarchies	L & H
Uniden	Uniform density of vertices in the layouts	G & F
Vert	Verticality of hierarchical structures	L & H

Table 2 – A summary of aesthetic criteria

Acronym	Constraint	Category
Centre	Place a set of given vertices in the centre of the layout	L & F
Dimens	Assign the dimensions of the symbols representing specific vertices	L & F
Extern	Place specified vertices on the external boundary of the layout	L & F
Neigh	Place close together a group of vertices	L & H
Shape	Layout a subgraph with a pre-specified shape	L & H
Stream	Place a sequence of vertices along a straight line	L & H

Table 3 - A summary of semantic constraints

Aesthetic	Tree	Planar	Hierarchy	Directed /Undirected
Area	✓	✓	✓	✓
Balan	✓		✓	
Bends		✓		
Convex				
Cross	✓		✓	✓
Degree				
Dim				
Length	✓		✓	✓
MaxCon	✓	✓	✓	✓
Symm	✓		✓	✓
Uniden				✓
Vert	✓		✓	

Table 4 - Showing which aesthetics apply to which class of graph

In two studies Purchase ([138] and [136]) assessed five aesthetics. The aesthetics were: the number of bends, number of edge crossings, angle of incidence of edges, orthogonality and symmetry. She finds that the number of edge crossings is by far the most important aesthetic. Bends and symmetry have a lesser effect, and maximising the minimum angle and maximising orthogonality have no significant effect at all. This justifies why all algorithms found try to minimize or remove the edge crossings. Batini et al. [8] found that 70 percent of the layouts had no crossings at all and the remaining 30 percent had an average of 8.65 crossings per diagram.

In a study of social science graphs (known as social networks) by Blythe, McGrapth and Krackhardt [15] it was found that structural and spatial factors influence individuals' perception of prominence. It is found that moving a vertex away from the centre decreases its perceived prominence. This suggests that centring important vertices in a graph leads to increased understanding of the graph. Batini et al. [8] reports that 40 percent of diagrams present emphasis on a special object. Of these 70 percent have the most important object in the centre and 20 percent have the important feature

emphasised by being larger and 10 percent show it being a different shape. They also conclude that is was important to minimize the number of symbols used and that grouping of symbols is important.

Batini et al. [8] also found that minimising both the edge length and bend in edges is an important feature of a diagram. 30 percent have no bends, and of the remaining 70 percent the average has 9.42 bends per diagram and 0.8 bends per edge. Minimising the area considered is another important feature.

In a study of what makes a diagram visually look good Ding and Matei [46] listed nine factors; these are: -

- visual complexity
- regularity
- symmetry
- consistency
- modularity
- sizes
- shapes
- separation, and
- traditional ways of laying out diagrams.

Ambiguity, recognizability and geometrical complexity affect the visual complexity of a diagram. A diagram is recognisable if it is physically recognisable or semantically recognisable. It is physically recognisable if the diagram is not too big or small. Such items as the edge lengths in a layout affect this. A diagram is semantically recognisable if its various interconnection properties are recognized. For instance, two diagrams could be laid out together, and the links make it seem as if they were one (see Figure 13 for example).

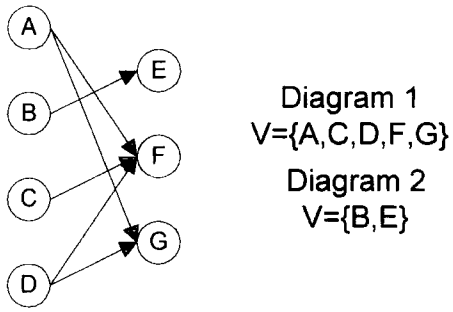


Figure 13 - A graph that is not semantically recognisable

If all the vertices of a diagram are related by some mathematical relationship, then regularity plays an important role in the layout of the diagram. The mathematical relationship can then be used to ensure that all the vertices are laid out in the same sort of manner. All the vertices can be laid out by using the mathematical relationship.

Symmetry is a kind of regularity. It is the "*correspondence in size, form, shape and arrangement of figure elements on opposite sides of a plane, line or point*" [46]. It is deemed to make a diagram more recognisable and beautiful.

A study by Dengler and Cowan [41] shows that how the vertices are ordered affects the understanding of the diagram. They show that if the vertices are positioned symmetrically, in a circle, grid or line then it is interpreted as having properties in common, or being equal in status. If the vertices are positioned centrally or nearer the top then they have special properties or have a higher status. A linear arrangement of vertices means that there is a sequence of information. This suggests that program comprehension is aided by common structures being easily detected in the layout.

Consistency is where structures that are the same in a graph are drawn with the same layout everywhere. For instance in a binary tree diagram, it is important that the left branch looks the same as the right branch.

Modularity in a diagram means that a diagram is made up from sub-diagrams of a standard plan or pattern. Therefore the reader does not have to learn new structures. The attributes of size, separation and shapes of a diagram are important as well. If a

diagram is too big it can be complicated to understand, whereas if it is too small it can be difficult to read. Line separation influences how well each line is identified and followed, and thus make it easier to read. Standard shapes should be used and the number of different shapes be kept to a minimum. People understand diagrams when they are laid out in a familiar way; this means that traditional methods should be employed when laying out a diagram.

2.5 Metrics

Earlier in this chapter it was stated that there are several problems with graph layout; one being what constitutes a 'good layout'. In order to define a 'good' layout it is necessary to measure the quality of the layouts. This is accomplished using metrics. In the following section the term metric is defined and some metrics used to measure the quality of graphs are given.

2.5.1 Definition

Measurement lies at the heart of many everyday events. Economical measurements determine price and pay increases. Measurements in atmospheric systems are the basis for weather prediction. Without measurement technology cannot function. But how is measurement defined? Fenton [57] defines it as: -

"The process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules."

Measurement is central to the evaluation of layouts. This is best represented by the IEEE [1] definition of a metric: -

"A quantitative measure of the degree to which a system, component or process possesses a given attribute."

So a metric in terms of graphs is a quantitative measure to which the graph possesses a given attribute.

2.5.2 Graph Metrics

No study has produced a complete list of metrics for graph layout. A recent incomplete study is by Purchase [139]. There have been several studies of graph layout algorithms ([44], [23] and [45]). A summary of the metrics used to evaluate them is shown in Table 5. The effect on the understanding of the graph is discussed earlier in the aesthetic section.

Metric	Meaning
Area	Area of the smallest rectangle with horizontal and vertical sides covering the layout
Cross	The total number of edge crossings
TotalBends	Total number of bends
TotalEdgeLen	Total edge length
MaxEdgeBends	Maximum number of bends on any edge
MaxEdgeLen	Maximum length of any edge
UnifBends	Standard deviation of the number of edge-bends
Uniflen	Standard deviation of the edge length
ScreenRatio	Deviation from the optimal aspect ratio, computed as the difference between the width/height ration of the best of the two possible orientations (Portrait and landscape) of the layout and the standard 4/3 ratio of a screen
ResFactor	Inverse of the minimum difference between two vertices, or two edge-crossings, or an edge crossings and a vertex
Symmetry	A number given to the symmetry of a graph
Cluster	The number of groups of vertices in the graph

Table 5 - Common metrics applied to graphs

In a study by Purchase [137] it is shown that the performance of the graph layout algorithm is quantifiable by measuring the understanding of the graph by a subject. After a subject has studied the graph for a length of time the subject's understanding is measured by asking a question about the graph such as "*what was the shortest route between two vertices?*" If the questions are all answered correctly then the layout is a successful, or poor if an incorrect answer is given.

2.5.3 Metric Calculation

In the section below methods of calculating some of these metrics are given. They are used later in calculating the quality of the graphs produced by the ANHOF system of call graph layout.

2.5.3.1 Number of Clusters

Organising data into sensible groupings is one of the most fundamental modes of understanding and learning. For example in program comprehension it has been perceived that in order to aid the understanding of a program's calling information all the calling information about a procedure should be grouped together in the resulting graph layout. This practice is common in many fields of science. It has therefore been the subject of much research, especially in the field of statistical analysis where it is known as Cluster Analysis.

Jain and Dubes [84] describe cluster analysis as the, "*formal study of algorithms and methods for grouping or classifying object*", where an object is either a set of measurements or relationships between the objects. It differs from discriminate analysis or pattern recognition because it does not use category labels that tag objects with prior identifiers. The object is to find a convenient and valid organization of the data.

Everitt [53] describes a cluster as, "*a set of entities which are alike, and entities from different clusters are not alike*". This definition is of little use when trying to measure the number of clusters in a graph. A graph consists of vertices that have positions on a plane and edges. It is therefore only possible to cluster data on the distance between the vertices. A better definition is again provided by Everitt [53] and is, "*A cluster is aggregation of points in the test space such that the distance between any two points in the cluster is less than the distance between any point in the cluster and any point not in it.*"

It is very easy to recognize a cluster of vertices if we see it, however it is very difficult to formalise how it is done and provide an operational definition of a cluster. Once a

definition of a cluster is set, automating the process offers distinct advantages over performing it manually, the main one being that it is applied consistently over any data.

A cluster is a subgraph and will have various properties, which can also be clustered. Jain and Dubes [84] suggest five types of information to cluster in a subgraph. These are: -

- **vertex connectivity** – the vertex connectivity of a subgraph is the largest number n such that at least n paths having no vertices in common join all pairs of vertices.
- **edge connectivity** - the edge connectivity of a subgraph is the largest number n such that at least n paths having no edges in common join all pairs of vertices.
- **vertex degree** – the degree of a connected subgraph is the largest integer n such that each vertex has at least n incident edges.
- **diameter** – the distance between two vertices, where the distance is given by number of the edges in the shortest path joining the vertices.
- **radius** – the radius of a connected subgraph is the smallest integer n such that at least one node is within a distance n of all other vertices in the subgraph.

The simplest method and the most common method of calculating the number of clusters of a graph is that of the distance between two vertices, a variation of the radius method suggested above.

According to Everitt [53] cluster analysis techniques can themselves be 'classified' into types, these are as follows: -

1. **hierarchical techniques** – this is where the clusters themselves are clustered together until they form a tree.
2. **optimisation-partitioning techniques** – in which the clusters are formed by optimisation of a 'clustering criterion'. The clusters are mutually exclusive, thus forming a partition of the set of entities.

3. **density of mode-seeking techniques** - in which searching for regions containing a relatively dense concentration of entities forms clusters forms clusters,
4. **clumping techniques** – in which the clusters or clumps can overlap.
5. **others** - methods that do not fall clearly into any of the four previous groups.

A good summary of all the above techniques can be found in [53], [54] and [84]. Many software engineering graphs are largely hierarchical in nature, clusters are therefore likely to be on the various levels, and even likely to be the leaf vertices of all the branches. Method four is therefore unsuitable. No part of a graph will be less densely populated than another, thus making method three unsuitable. Method two and five are unsuitable because they were largely more complex than needed. It was found by experimentation that hierarchical techniques (method 1) provide a valid clustering technique.

2.5.3.1.1 Hierarchical Clustering

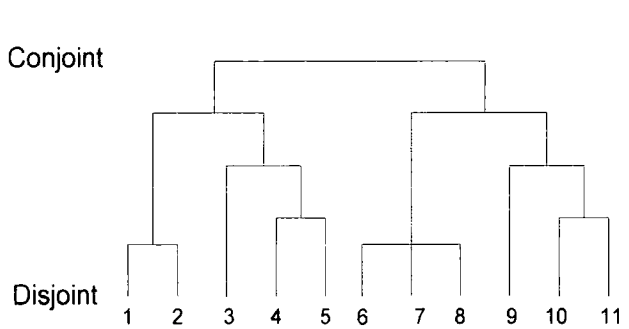
There are many methods of Hierarchical Clustering. A few are: -

- nearest neighbour or single link method,
- furthest neighbour or complete link method,
- centroid cluster analysis,
- median cluster analysis,
- group average method, and
- ward's method.

To describe each method here is unnecessary. A good description is given both in [84] and [53]. Various statistical packages implement each method and a summary can be found in [54]. One such package is SPSS, which has implementations of all the above methods. However none of the methods provide a number of clusters of the data it is processing. The methods are simply a method of clustering the data. This clustering is represented as a Dendrogram. A Dendrogram is a special type of tree structure that provides a convenient representation of a hierarchical clustering. A Dendrogram

consists of layers of vertices, each representing a cluster. Lines connect vertices representing clusters, which are nested into one another. Cutting a Dendrogram horizontally creates a clustering. Figure 14 shows a simple Dendrogram and the cuts that produce the clustering. The number of clusters in that clustering is obtained by calculating its number of members. The Dendrogram is not taken from any specific graph. It is merely an example Dendrogram.

In this research the number of clusters in a graph will be calculated by obtaining a clustering at level three of the Dendrogram, the level that is two up from the base of the Dendrogram. Also where most of the objects are involved in a cluster, but it is still possible that they are on their own. The size of the clustering is then calculated to obtain the number of clusters in the graph. For instance at level three in Figure 14 there are six clusters. If Figure 14 represented a graph then there would be six clusters in it.



Level	Clustering
6	{(1,2,3,4,5,6,7,8,9,10,11)}
5	{(1,2,3,4,5),(6,7,8,9,10,11)}
4	{(1,2),(3,4,5),(6,7,8),(9,10,11)}
3	{(1,2),(3),(4,5),(6,7,8),(9),(10,11)}
2	{(1,2),(3),(4),(5),(6,7,8),(9),(10),(11)}
1	{(1),(2),(3),(4),(5),(6),(7),(8),(9),(10)}

Figure 14 - A simple Dendrogram and its clustering

An example of each method of clustering will now be given. In order to compare the various clustering methods an example graph is given in Figure 15. Although simple this example begins to show some of the differences between the various methods. A comparison is also given in [84] and [53].

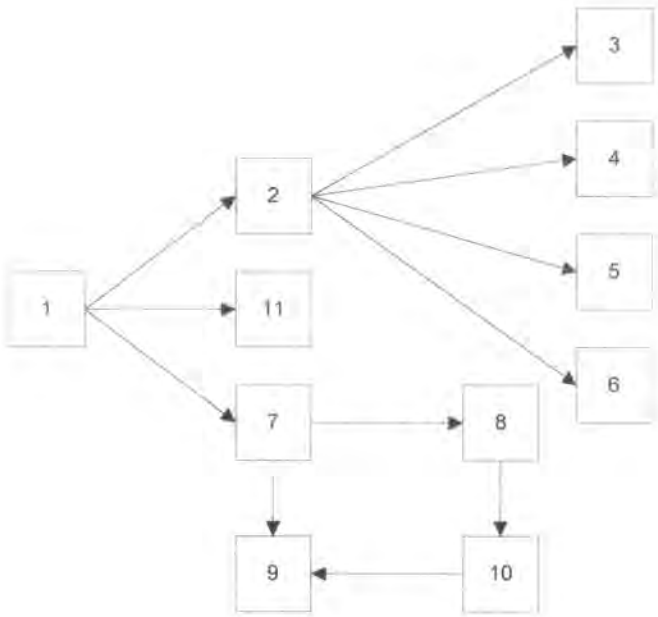


Figure 15 - An example graph to cluster

The X and Y positions of the vertices are entered into SPSS. This then performs any of the necessary calculations of the analysis method. Table 6 shows the results of running the graph through SPSS and reading the resulting Dendrogram at level three

Method	Clustering	Number of Clusters
Centroid	{(6,5,8),(3,4),(9),(10),(1),(7,11),(2)}	7
Median	{(7,11,2),(1),(3,4),(6,5,8),(10),(9)}	6
Furthest Neighbour	{(7,11,2),(1),(3,4),(6,5,8),(10,9)}	5
Average Linkage (Within group)	{(7,11,2),(9),(1),(3,4),(6,5,8),(10)}	6
Nearest Neighbour	{(7,11,2),(9),(3,4,5,6),(8),(10),{11)}	6
Ward	{(5,6,10,8),(3,4),(1),(9),(7,11,2)}	5
Average Linkage (Between Groups)	{(6,5,8),(3,4),(9),(10),(1),(2),(7,11)}	7
Manually	{(1),(2,7,11),(9),(10),(3,4,5,6), (8)}	6

Table 6 - The clustering of graph in Figure 15

Table 6 shows that three methods produce the same number of clusters as the manually procedure. However they all, except nearest 'neighbour analysis', fail to cluster vertices labelled 3,4,5 and 6 together. Instead, clustering together, in some form, vertices 3,4,5,6 and 8, usually as two clusters. They cluster across levels of the graph. This is a common

outcome when the methods are applied to other graphs. When performing clustering manually, very few clusters traverse across the levels of a graph. They cluster vertices that are close together. These are generally on the same level. The problem lies in defining the proximity of vertices and there is advantages were the computer performs the clustering automatically. The distance was unknown to the user, but was standardised by the computer's implementation.

When comparing the output of the above methods it becomes obvious that most clusters are not only on the same level but move across levels of the graph. The data that is used in the method is largely clustered already and therefore the clusters are present at an early stage of the clustering process. Everitt [53] suggests that nearest neighbour analysis gives rise to what is known as 'chaining' of clusters, where the clusters are produced at a relatively low level of objects. This is another reason for choosing this method. In the above graph and others, nearest neighbour analysis continually matches the clustering gained manually. For this reason nearest neighbour analysis is used to calculate the number of clusters in a graph.

2.5.3.2 Edge crossings

Call graphs are hierarchical in nature. A number of methods can be used to calculate the number of crossings in a graph. One is to use the interconnection matrix. This is a matrix built up of all the connection between vertices of two levels in the hierarchy. These are best explained by Warfield [168] and Sugiyama et al. [159]. From these interconnection matrices the number of crossings can be calculated. There are various methods of doing this and are summarised in [159].

For each level on a graph an interconnection matrix is required. Applying the above methods is time consuming and complicated. A better method is to use the mathematical equation of the edge. In the ANHOF method an edge is a straight line and consequently has a mathematical equation. Whereas an edge that is on a grid based system does not have such a simple equation. In order to calculate if two edges cross it is necessary to know the mathematical equation of the edges. The layout algorithm is design to work on graphs laid out on a two dimensional plane. Where simple geometry obtains the

equation of the edge. A simple guide is available in many books, one such is [135]. It is necessary to know two points on the line; these are the points that the edge connects on the destination and departing vertices. The equation of the edge is then given by formula given in Equation 1.

$$0 = \frac{Y_2 - Y_1}{X_2 - X_1}(X - X_1) - Y + Y_1$$

Where: X_1 is the x coordinate of the departing point
Y_1 is the y coordinate of the departing point
X_2 is the x coordinate of the destination point
Y_2 is the y coordinate of the destination point

Equation 1 - The equation of an edge

The process of calculating if two lines cross is given by Algorithm 1. To calculate the number of crossings in a graph Algorithm 1 should be performed with each edge cross against every other edge.

Given four points (X_1,Y_1), (X_2,Y_2), (X_3,Y_3) and (X_4,Y_4) on two edges where $X_1<X_2$ and $Y_2<Y_1$ and $X_4<X_3$ and $Y_3<Y_4$. The process to find if the two edges cross is given in Algorithm 1, the point where the two edges cross is (X,Y): -

1. Obtain the departing (X_1, Y1) and destination (X_2, Y2) coordinates of edge 1.
2. Calculate the equation of edge 1 by using Equation 1.
3. Obtain the departing (X_3, Y3) and destination (X_4, Y4) coordinates of edge 2.
4. Calculate the equation of edge 2 by using Equation 1.
5. Calculate Y coordinate of the intercept by solving
   equation of edge 1 = equation of edge 2.
6. Calculate X coordinate of the intercept by substituting the value for Y, found in step 5, back into equation of edge 1.
7. If the point (X, Y) is in the range given below then the lines cross.

$$\text{Min}(X_1,X_4) \leq X \leq \text{Max}(X_3,X_2) \text{ and } \text{Min}(Y_1,Y_3) \leq Y \leq \text{Max}(Y_2,Y_4)$$

8. Return (X,Y).

Algorithm 1 -Showing when two lines cross

2.6 Summary

The chapter above has presented the current research in the diverse field of automatic layout of graphs. It has presented automatic layout algorithms for the general graph classes of directed and undirected graphs, and the classes of trees, planar graphs, and hierarchies. It has also presented the areas of graph metrics and aesthetics, defining them, giving examples of them and providing a discussion on their use.

Automatic layout of graphs is a difficult area. There are many factors to take account of some are given in the chapter above. It is a field that is worth researching. There is still much research to do. It has shown that it is difficult to describe what constitutes "good" graph. This is because it is a subjective area. However the quality of graphs can be measured using metrics. This chapter has discussed how they can be used and a suggested method of calculating them. Current layout algorithms cause many of these metrics to be uncontrollable. The metric performance of these layout algorithms cannot be improved unless they are tailed to specific graphs. Therefore the required aesthetics can be clearly stated and the performance can be controlled.

In the next chapter example software engineering graphs are given. Together with a summary of the areas of graph layout grammars, graph representation languages and graph display tools. Algorithms are presented for graph and subgraph isomorphism.

3. Some Graph Uses

Chapter 2 discusses the diverse area of automatic graph layout. It shows that graph theory can be used to define many classes of graphs and as the basis to describe the process of automatically laying them out. There are many more domain specific graphs that call upon this graph theory and methods of describing the software engineering graphs in terms of a simple language. Graphs are a visualization technique and this representation technique requires a method of displaying them either on paper, or on a computer display screen using a graph display tool.

In software engineering, graphs are used as a method of representing information about a program. For instance its structure or how the program processes data or how data flows through the graph. In modern software engineering, graphs are increasingly used as a documentation, comprehension and planning aid. In this chapter some of the graphs that are used as program comprehension aids in software engineering are defined. A preview into graph specification languages is also given. Then a summary and survey of graph tools is given.

Graphs, and in particular, labelled graphs are a very powerful and universal tool that are widely used in computer applications. One of the most important problems in graphs is the comparison of graphs with each other. This is called graph isomorphism. Two graphs, G_1 and G_2, may be matched using bijective mapping between the vertices of G_1 and G_2 such that the structure of the edges is preserved by the mapping function. The existence of a mapping defines that G_1 and G_2 are isomorphic.

The graph isomorphism problem has been the subject of much research over the years. It is still not know whether it is NP or P hard. This chapter of the thesis discusses the isomorphism problem, and discusses the issue of which complexity class is applicable. Examples of matching algorithms are given, along with a comparison.

3.1 Software Engineering Domain Graphs

This section defines some of the graphs that are used in the domain of software engineering to represent the software's structure. It will provide formal definitions of the graphs and give the symbols that are used to draw them. It describes common structures of the graphs. These graphs will be restricted to two dimensional (2D) directed graphs and in particular to the following graphs types: -

- call graphs,
- flowcharts,
- control flow graphs, and
- data flow diagrams .

The above graphs are restricted to those that can be drawn from sequential, medium to high level languages, those that are not too close to machine code, such as C, and those that do not allow parallel computation or inheritance.

3.1.1 Call Graphs

A call graph is used in software engineering to represent the calling relationships between procedures. Jeffries [85] suggests that in order to understand the whole program, when comprehending the program, programmers have to read the program code first and they use a common method. They read it in the order in which it would be executed, main procedure first, then procedures called by the main procedure, and then procedures called by those procedures, etc. This knowledge is represented by a call graph and represents a top down approach to program comprehension. Another justification for call graphs is given by Smith [153] who says that, "*a program's global structure is seen distinctly when its routines are connected by their call to one other.*"

In this thesis the concept of a procedure is used in a call graph. In the C language procedures are called functions. The term procedure will be used as a generic term that represents a sub unit of a program. When one procedure calls another, a hierarchical

structure becomes apparent. Hierarchical layout algorithms therefore provide the best method of automating the layout of them. More formally, Ryder [145] suggests that a call graph could be represented by a directed graph G = (N, E) where: -

- *There exists a procedure vector set such that for each procedure P_i defined with m procedure parameters has a set of m-tuples of external procedures, which, during the execution of the program can be associated with its m-tuple of procedure parameters.*
- *Each vertex N_i corresponds in a one to one manner to a procedure P_i and its procedure vector set.*
- *If P_i contains a reference $B_o(B_1,..., B_k)$ then for each expansion P_{j0} $(P_{j1},.... P_{jk})$ of that reference there is a directed edge (N_i, N_{j0}) in the graph and $(P_{j1},...., P_{jk})$ is in the procedure vector set of N_{j0}.*

According to Grove, DeFouw, Dean and Chambers [74] there are two types of call graph, context insensitive and context sensitive, a definition and discussion is given below.

3.1.1.1 Context Insensitive Call Graphs

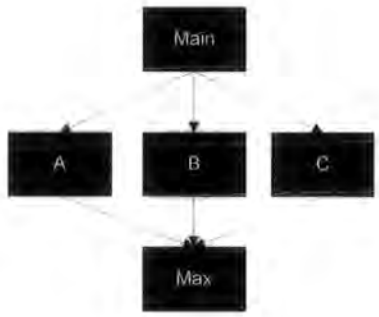
Each procedure is represented by a single vertex in the graph, each vertex has an indexed set of call sites and each call site is the source of zero or more edges to other vertices representing callers of that site [74]. An example can be found in Figure 16. In this figure there is no distinction between the types of parameters used in procedures A and B when calling 'max'. In the graph (Figure 16(b)) they call the same instance of procedure 'max' and this is said to be context insensitive.

```
Procedure Main()
{return A() + B() + C()}

Procedure A()
{return max(4,7)}

Procedure B()
{return max(4.5,2.5)}

Procedure C()
{return max(3,1)}
```



(a) code (b) call graph

Figure 16 - (b) A context insensitive call graph and (a) its corresponding code

3.1.1.2 Context Sensitive Call Graphs

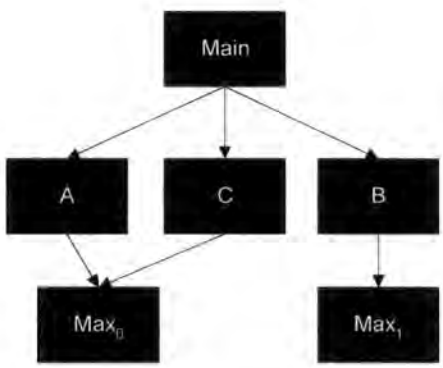
A procedure may be analysed separately for different calling concepts. Each concept is called a contour [74]. An example can be found in Figure 17. In this figure procedure B calls 'max' with floating point parameters and procedure A calls 'max' with integer parameters. Each of these types of parameters is dealt by different instances of procedure max, and the call graph (Figure 17(b)) represents this and is therefore said to be context sensitive.

```
Procedure Main()
{return A() + B() + C()}

Procedure A()
{return max(4,7)}

Procedure B()
{return max(4.5,2.5)}

Procedure C()
{return max(3,1)}
```



(a) code (b) call graph

Figure 17 –(b) A context sensitive call graph and (a) its corresponding code

Call graphs have been used in many projects in Durham, such as the AMES project [19]. The symbols used when drawing call graphs in these projects and others are given in Figure 18.

Figure 18 - The symbols used in call graphs

3.1.1.3 Aesthetics

Call graphs have a natural hierarchical structure. Therefore they are laid out using aesthetics similar to those used in hierarchical graphs. These are: -

- minimize the area,
- balance the graph in terms of its horizontal and vertical axis to match the media that it is being displayed on,
- minimize the edge crossings and edge length, and
- centre father vertices over their sons.

3.1.1.4 Example

The call graph of the program 'Lines.C' (listed in the appendix 1) is shown in Figure 19. It shows that procedure "qsort" is a recursive function and also shows how it is represented. In Figure 19 there are no edge crossings and edge length has been kept to a minimum. The parent vertices are all centred over their children and the graph prints well on printed media.

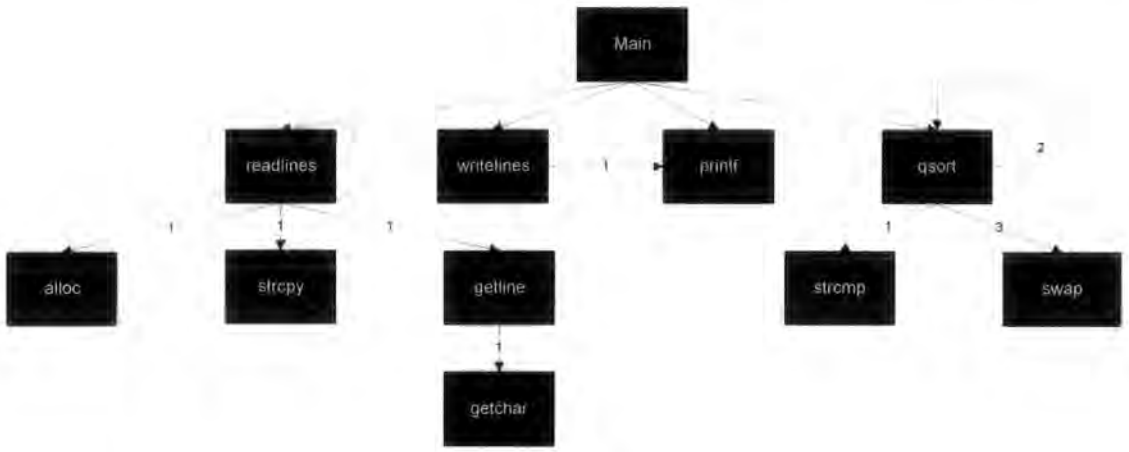


Figure 19 - The call graph of the 'Lines.C' program

3.1.1.5 Common Structures

When graphs are laid out manually, the parts of the graph that are recognizable and are laid out first if a method of laying them out is known. This leads to certain common structures becoming obvious in the final layouts. These can then be used to aid the understanding of the graph and aid automatic layout of the graphs. Earlier work at the University of Durham, summarised in Munro, Burd, Chan, and Young [117], has discovered that there are various common structures in call graphs. These structures are discussed in Chapter 4 of this thesis as they form the basis of the ANHOF method of call graph layout.

3.1.2 Flowcharts

A flowchart is used to design a program and to describe how the program performs its task; again it has a hierarchical structure. There is no definitive definition, the following is compiled from several sources and is: -

A flowchart is a means of portraying, in graphic form, a sequence of specified operations performed on identified data. This is usually drawn using standard symbols.

This type of diagram was first used in 1946-1947 by Goldstein and von Newmann [72]. In the 1960's various organisations defined standard symbols. In the software

engineering domain the ANSI X3.5 [5] symbols are used (Figure 20). The symbols cover a variety of areas including operations, storage devices and input /output methods.

There are two types of flowcharts: -

1. system chart - this describes the sequences of data handling, identifying the input and output sources.
2. flow diagrams (charts) - this describes the data handling, it explains each of the steps involved in each process. These are the most common type.

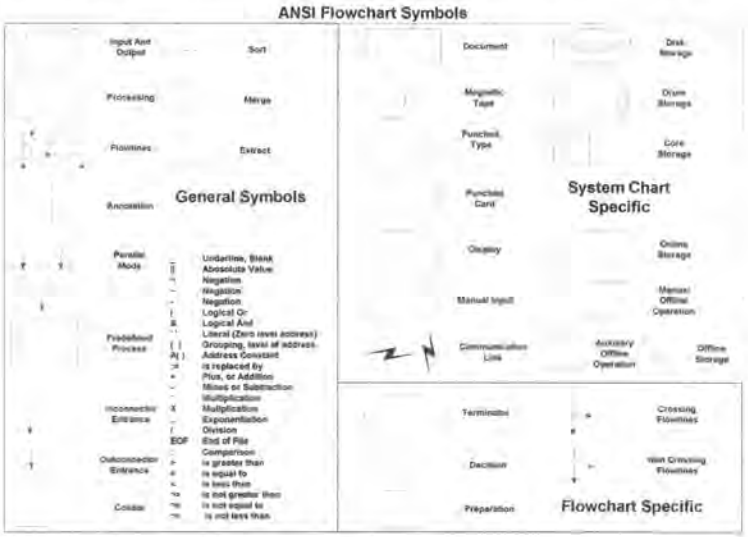


Figure 20 - The ANSI flowchart symbols [31]

Flowcharts are used as a design tool. A survey by Schneiderman, Meyer, McKay and Heller [151] of 45 Fortran texts shows that 14 of them employ flowcharts extensively and 19 use them occasionally. The remaining 12 do not use flowcharts. Flowcharts are heavily used as teaching aids and as a simple comprehension aid. They tend to get very large for large programs however.

3.1.2.1 Aesthetics

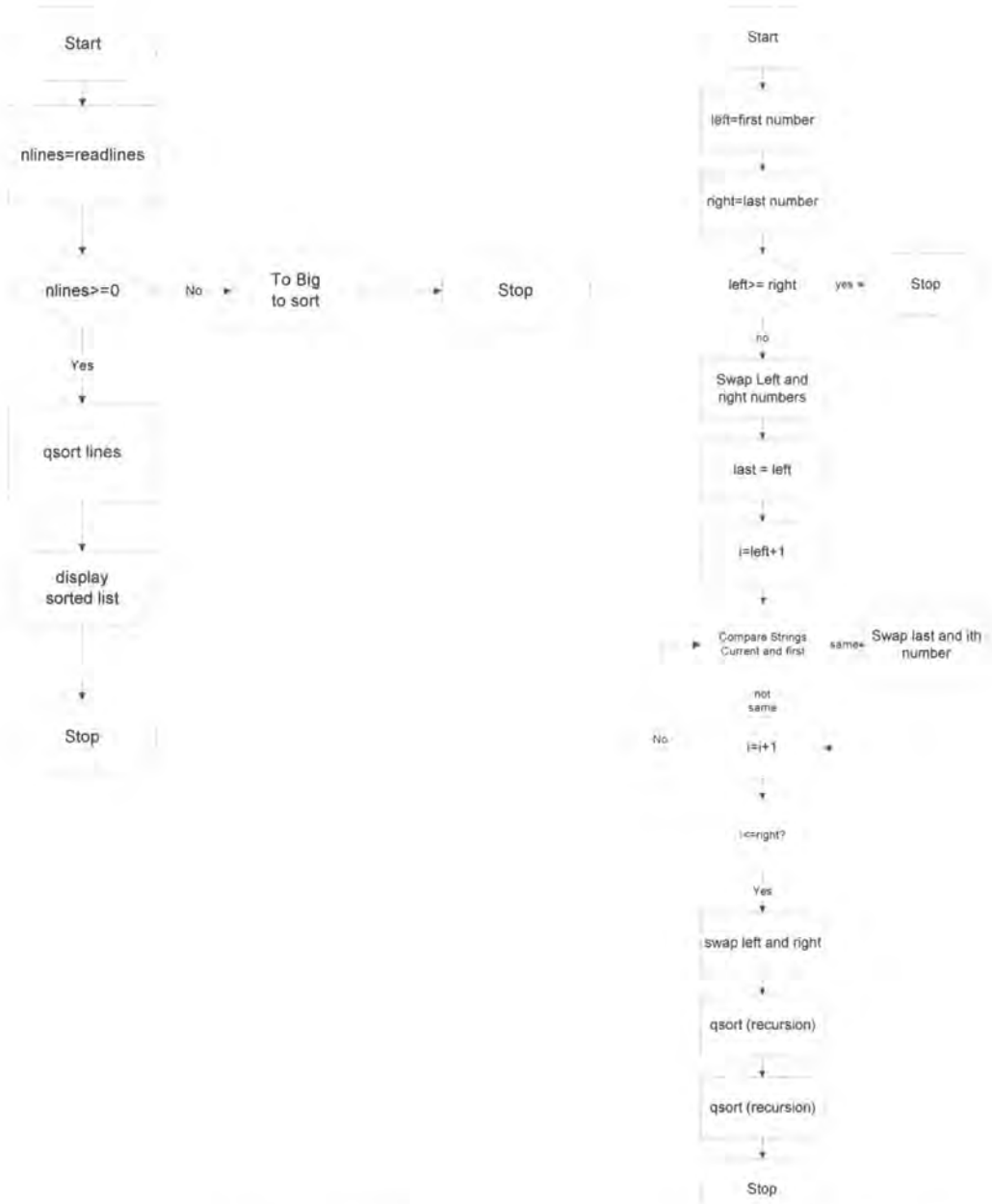
These graphs will generally be long and thin, and as a consequence they will be difficult to display. Aesthetics will be very important here in order to ease understanding. They are commonly laid out following the aesthetics below: -

- minimize the area,
- minimize the edge length and edges crossings, and

- the flow of the graph should be down one of the axis of the display media.

3.1.2.2 Example

The flowchart in Figure 21 shows the structure of the main functions in the program 'Lines.C', the entry function ('main') and function 'qsort'. The figure also provides a method of representing recursion. The graphs all flow naturally down the page with no crossings present and the edges are short and do not cross.



(a) Main function (b) Procedure qsort

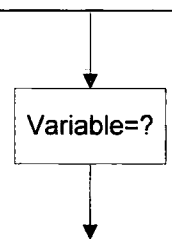
Figure 21 - The flowchart of 'Lines.C'

3.1.2.3 Common Structures

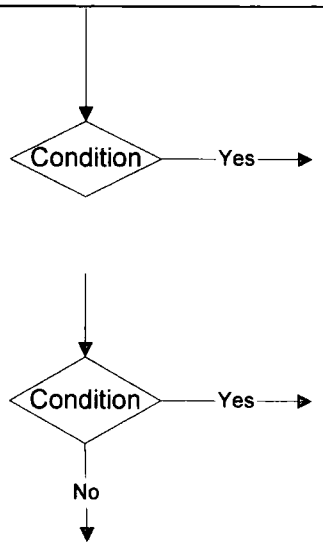
Flowcharts tend to be an abstract view of the program code. Therefore the common structures are the same as the common operations in the programming language. Analysis of several programming textbooks, such as [93], [37] and [51], suggests that the common structures used in programming languages and hence in flowcharts are the following: -

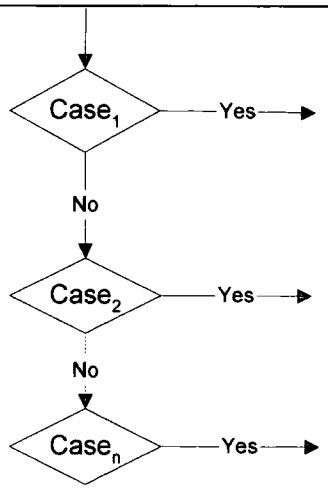
- variable assignment,
- conditional statements,
- branching statements, such as switch,
- while loops (loops where the exit clause is the first statement),
- continuous loops (loops where the exit clause is in amongst the other steps), and
- do until loops (loops where the exit clause is the last statement).
- For Loops (Repeat the steps a set number of times).

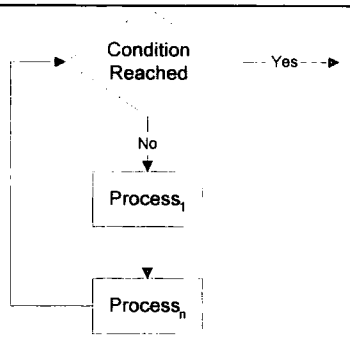
In Table 7 example code of each of these common structures will be given, together with the flowchart representation of them: -

Type	Example Code	Flowchart Representation
Variable Assignment	Begin X: =0 End	

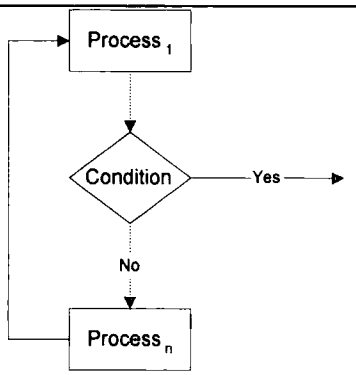
Type	Example Code	Flowchart Representation

Conditional Statements	Begin
	If (x=1) then
	do something
	End
	
	Begin
	If (x=1) then
	do something
	else
	do something
	End
		

Branching Statements	Begin case of x do something do something End	

While Loops	Begin
	While (x<>1)
	Begin
	do something
	do something
	End
	End
		

Type	Example Code	Flowchart Representation

Continuous Loops	Begin
	Repeat
	Begin
	do something
	do something
	if (x=1) then exit
	End
	End
		

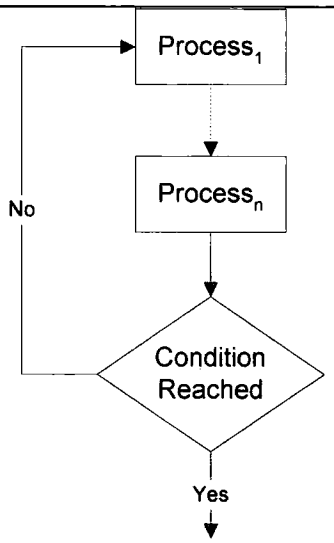
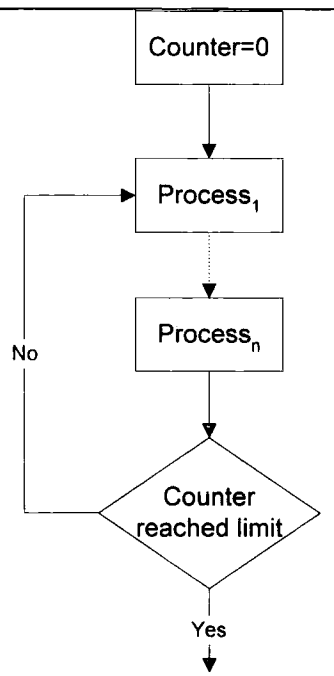
Do Until Loops	Begin Do do something do something Until (x=1)	
For Loops	Begin for x=1 to10 Begin do something End End	

Table 7 - The common structures of a flowchart

3.1.3 Control Flow Graphs

Control flow graphs are similar to a Flowchart, except the graph shows only the flow from one step to another. It does not detail what happens at each step. Johnson, Pearson and Pingali [86] provide a good definition: -

"A control flow graph is a graph with two distinguished vertices, known as start vertex and end vertex. Such that every vertex occurs on some path from the start to the end. The start vertex has no predecessors and the end vertex has no successors."

Control flow graphs are generally used as a simpler form of flowcharts. They have been used extensively in software metrics, e.g. in measuring the complexity of a system or program [109] or as a measure of the structure in a program [59].

The symbols used in a control flow graph are given below.

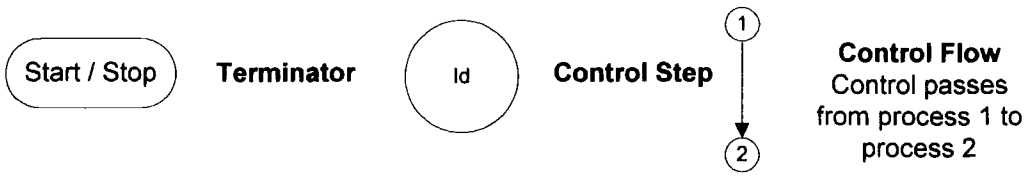


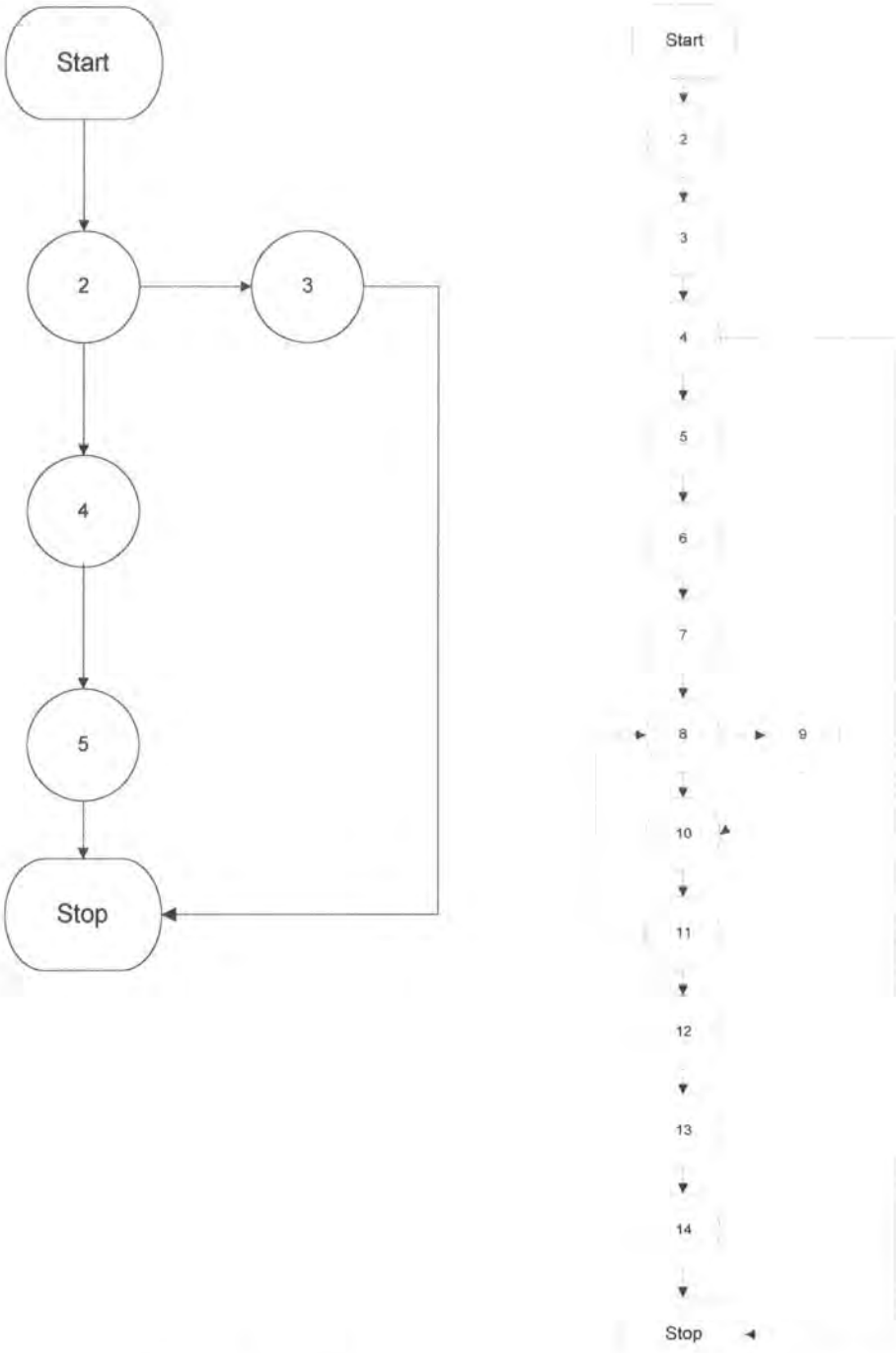
Figure 22 - The symbols used in a control flow graph

3.1.3.1 Aesthetics

Control Flow Graphs are similar to flowcharts and are therefore drawn in similar manner. They are laid out with the same aesthetics as flowcharts.

3.1.3.2 Example

Figure 23 shows the control flow graph representation of the main function and function 'qsort'. This is similar to the flowchart, except the information about each step is omitted.

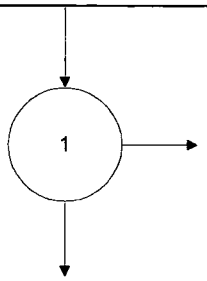
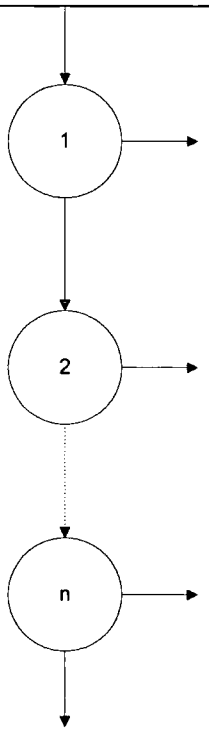


(a) Main Procedure (b) Procedure qsort

Figure 23 - The control flow graph of the 'Lines.C'

3.1.3.3 Common Structures

The control flow graph is another example of an abstract view of the branching structure in the program. They therefore have the same common structures as flowcharts but are represented differently. The common structures are given in Table 8.

Operation	Control Flow Representation
Conditional Statement	
Branching Statements	

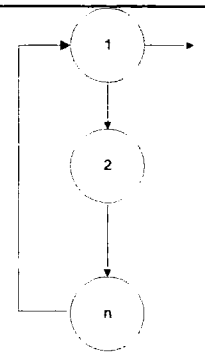
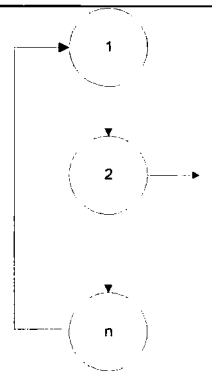
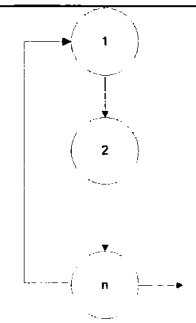
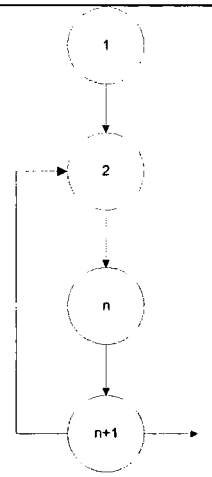
Operation	Control Flow Representation
While Loops	
Continuous Loops	
Do Until Loops	
For Loops	

Table 8 - The common structures of a control flow graph

3.1.3.4 Control Flow Graph Decomposition

Control flow graph decomposition is a method of breaking down the control flow graph into its basic graph structure. There has been some work on control flow graph decomposition, this has been summarised by Fenton and Pfleeger [58]. They suggest that control flow graphs consist of several primes. The primes are the common structures suggested above. They suggest that a control flow graph could be described in terms of the degree of its vertices. A vertex can have an "in" degree, the number of edges arriving at the vertex and an "out" degree, the number of edges leaving the vertex. This is the same as the 'fan in' and 'fan out' notion by Henry and Kafura [76] used throughout this thesis. Using these definitions a control flow graph can be described as: -

A control flow graph is a directed graph in which two vertices, the start vertex and the stop vertex, obey special properties: the stop vertex has a out-degree (fan out) of zero and the start vertex has a in-degree (fan in) of zero and every vertex lies on some path from the start vertex to the stop vertex. (Adapted from [58])

These primes are then combined together using sequencing and nesting to make up the control flow graph. Sequencing is where a prime follows on from another and nesting is where a prime is inserted in between a vertex and the next vertex, for more details see [58].

In the past it was believed that structured programs could be expressed in terms of three constructs; sequence, selection and iteration. This was a result of earlier work by Bohm and Jacopini [18]. However in modern languages such Ada and Modula-2 there are commands for performing the primes (common structures) above, therefore structured programs are expressed as sequence, selection, iteration and the primes above. Fenton [58] calls these S-graphs and concludes that if a control flow graph can be solely expressed in terms of these S- graphs then it is said to be well structured.

3.1.4 Data Flow Diagram

A general definition for a data flow diagram (DFD) is provided by the IEEE [1] and is: -

"A diagram that depicts data sources, data sinks, data storage, and processes performed on data as vertices, and logical flow of data as links between the vertices."

The symbols that will be used in these diagrams are given in Figure 24.

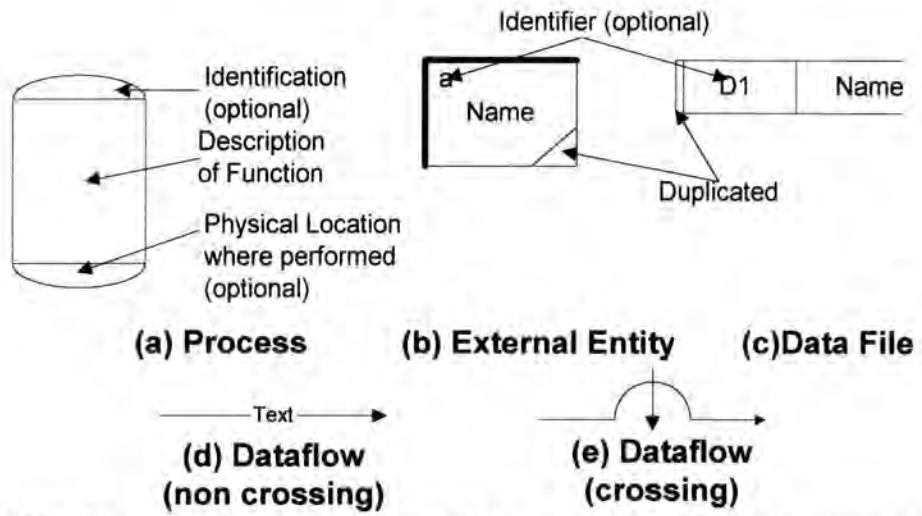


Figure 24 - The symbols used in a data flow diagram

The DFD may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels, which represent increasing information flow and functional detail. A level 0 DFD is also called a fundamental system model, representing the entire software element as a single process with the input and output represented as arrows from it. Additional processes are added as the level 0 DFD is further partitioned making the higher level graphs.

Data flow diagrams are extensible used in database systems. There are many texts that give examples of their use, including [181] and [66]. In addition there are many papers

that deal with systems that automate their extraction and drawing that give real world examples, two being [149] and [134].

3.1.4.1 Aesthesis

Generally DFDs are complicated with flow being spread all around the area of display. Aesthetics are difficult to apply because of this. However, if the following are applied then the graph may be more readable: -

- minimize the edge crossings and edge length, and
- balance the graph in terms of its horizontal and vertical axes.

3.1.4.2 Example

Gane [66] provides an example system analysis from which the data flow diagram can be drawn. This system analysis is found in the Appendix 1 and the resulting data flow diagram is given in Figure 25. This graph has many edge crossings, which are dealt with by the use of the symbol given in Figure 24(e). However the graph is generally easy to follow because of the use of identification numbers on the vertices that give it a reading order and something for the eye to look for.

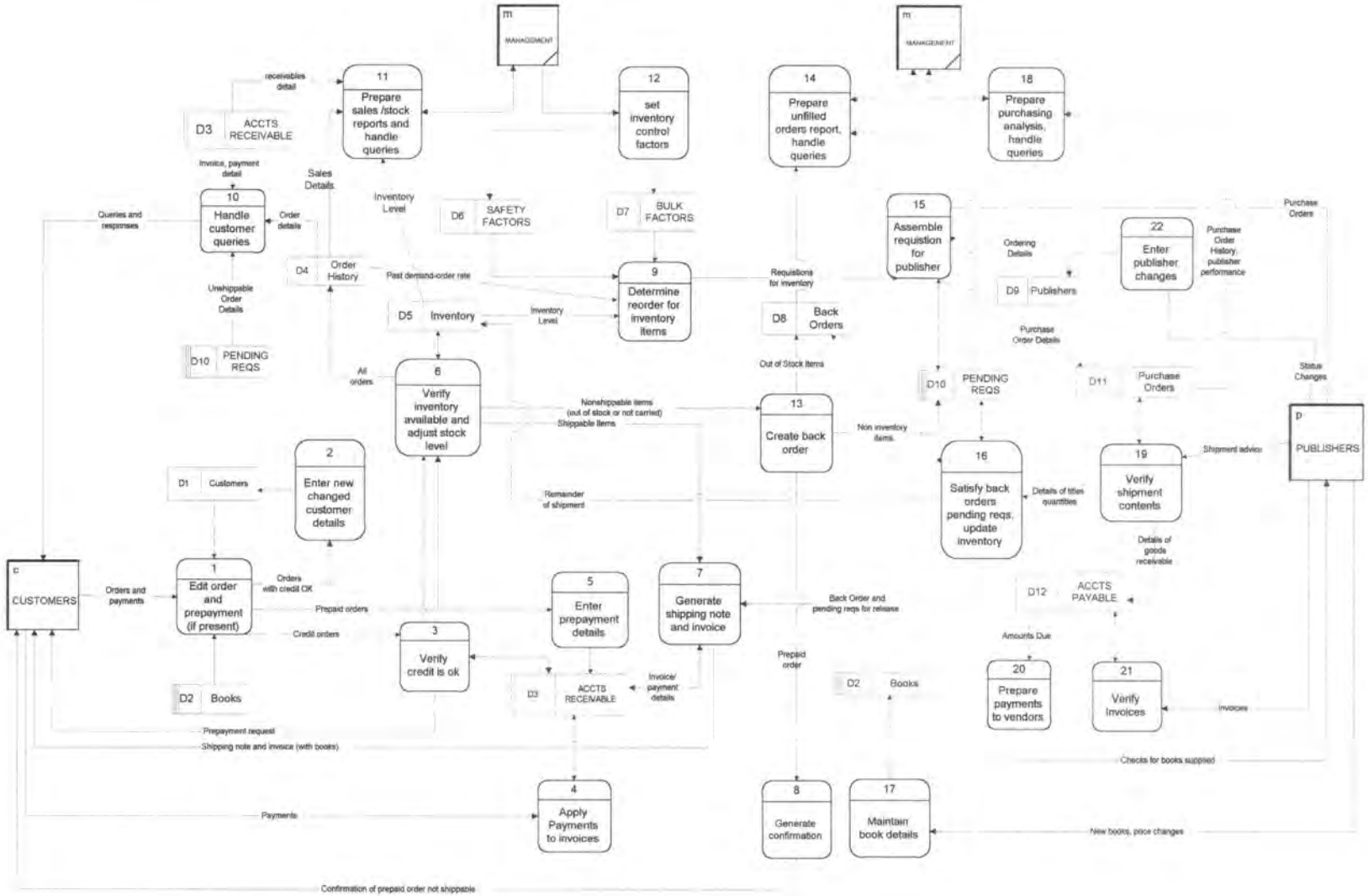


Figure 25 - An example of a data flow diagram

3.1.4.3 Common Structures

Very few common structures are found in data flow diagrams and published work on the topic could not be found. After a few data flow diagrams were laid out manually, common structures were found that generally involve two vertices and one edge and they represented the basic operations that can be performed in a data flow diagram. Consequently there will be many of them, but attaching a model layout to them will be hard.

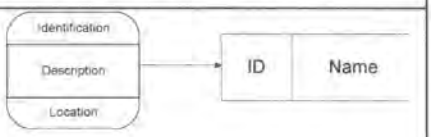
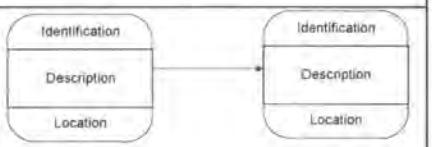
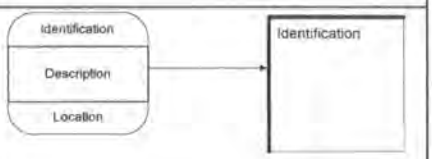
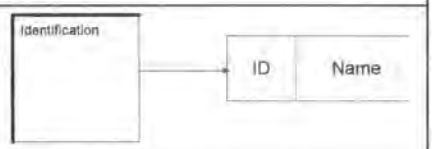
Name	Data Flow Representation
Process flows to database	
Process flows to process	
Process flows to entity	
Entity flows to database	

Table 9 - The common operations of a data flow diagram

3.2 Graph Specification Languages

Traditionally graph specification languages have been a method of describing a picture. An original language is PIC, developed by Kernighan [92]. It is a powerful language that provides a front end to TROFF for drawing simple pictures. Kernighan has adapted

it to layout diagrams in specific domains, for instance structure diagrams in chemistry. The language has been updated so that it provides a front end to ML [90]. It has also been made more powerful, using the features of ML to draw new types of diagrams, for instance pie charts. A very simple and largely accepted language is LOGO [122]. It is popular as it is taught in schools teaching pupils to understand basic mathematics and basic language concepts, e.g. procedures. It is a language that can be used to draw very simple line drawings.

Today graph specification languages are little more than file formats that are used to describe a graph so that a graph display tool can display it. File formats often provide tough problems both for the software engineers who write programs and people who are using them. Software Engineers want formats that store data in an efficient manner, and are easy to read and write. Users want a way to save their data in a convenient and fast manner, where they do not want to be concerned with the choice of a specific format.

The consequence is that almost every graphic or desktop publishing system has its own file format optimised for the needs of that product. This means that direct data exchange between different products is difficult since the file formats are often mutually exclusive. Most programs contain numerous converters that transform data between different formats.

Having converters is inconvenient for the user. First it means that n^2 converters are ideally needed to exchange data between n programs. However it is unlikely that each program can read and write each other's file format. Therefore it is inconvenient for the user, as he/she has to find a format that is common to both programs. Data may be lost by translating the file. One way to avoid the use of converters is to provide one powerful format that does everything, or, better, has a core part which is understood by all the participating applications, and can be easily extended to meet a particular application's needs.

A file format is not the type of file (binary or sequential) that is used to store information, but how certain information is stored within that file. This method of representing the information can be called a language. This language, because of its

very nature is domain specific. The file format therefore is used to store a program within this language.

In the following section some examples of file formats will be given, the advantages and disadvantages are discussed.

3.2.1 Languages

There are many different file formats (graph specification languages) available for graph drawing packages including a mixture of commercial and educational based formats. The educational languages tend to be ISO/ASCII files that are easy to edit and make by hand. The commercial ones tend to be secret and binary and are therefore more difficult to manipulate by hand or generated by other programs. It is for these reasons that in the following section academic languages are given that are stored in ASCII/ISO sequential files. In addition to the ones below there are dot (APPLE Research) and various Internet based formats such as VRML. Further information can be obtained from the graphic file format F.A.Q. [2] and other web pages. In the following section there is a discussion of the formats of: -

- adjacency Lists,
- daVinci,
- graph tool,
- graph modelling language (GML), and
- visualizing compiler graphs.

The manner in which the graph is represented as a data structure influences the ease in which certain types of graph are manipulated. When working with hierarchical graphs, a term based representation can be used such as that employed by daVinci. However, this representation has an implicit ordering of vertices in the graph and so makes cyclic graphs hard to manipulate. Many other systems use a variant of the GML method, which has two discrete sets, one for the vertices and the other for the edges. The edges have reference to a source and destination vertices. This allows more general graphs to be manipulated. However the connection between vertices are difficult to derive from inspection.

3.2.1.1 Adjacency Lists

An adjacency list is a file that contains every vertex in the graph and, following each one on the same line is a list of all the vertices it is connected to. Many systems use simple adjacency lists, perhaps enriched with labels or co-ordinates. Often the end of the line terminates an adjacency list. While this format is convenient and easy to use in these systems, it has several disadvantages. First it is difficult to expand because new vertices can be connected to existing ones, expansion therefore becomes more than just adding the vertices to the end of the list. Existing members may have to be modified. Second, labels are usually restricted to one character or a single word; often this is due to the maximum line length of a file. However this can be overcome. Further the degree of a vertex is limited on systems that do not support arbitrary line lengths, because the length of the line restricts the number of vertices that can be on that line.

3.2.1.2 daVinci

The daVinci language possesses many powerful features. It extends a basic list of vertices and edges to add many other features such as different vertex styles. It uses simple 'user friendly' keywords to represent graphs. These however are one letter; they would be better as whole words, e.g. edge instead of e. A large graph would be very difficult to understand because each vertex contains a set of other vertices it is connected to. The effect is to make it more difficult to structure the file properly, e.g. in getting the number of brackets correct. The language can only be used to describe a hierarchy. It could however be adapted to model general graphs. It does not allow each vertex to be given a position and this is quite important when an automatic graph layout algorithm is to be developed. It is because the daVinci file format is meant as a method of getting the graph into the drawing system and the layout is handled by the system when it is correctly stored. It could be adapted to perform this. Graph vertices are given locations by an automatic graph layout algorithm, which cannot be turned off. daVinci uses the Sugiyama et al. [159] automatic graph layout algorithm for laying out its graphs, details of its implementation can be found in [64]. It does however support a large number of features. The language is connected to a graph tool; which is a good,

stable and fast tool. If the proposed adaptation were made this tool could not be used. There are better languages available with tools connected to them.

The language is a user-friendly file format. It can easily be read and understood through reading the raw file. It, however, does not allow anything but hierarchies to be represented. Without proper layout these files quickly become unreadable. However, the style could be copied and modified to represent general graphs. The file format is represented as a grammar in Appendix 2.

3.2.1.3 Graph Tool

Graph Tool and its graph representation language were developed in Durham by Bodhuin [16] with later modifications by Young [180]. A Graph Tool file is frequently referred to in this thesis and it is to the Graph INformation file (a GIN File) that this refers and not to the two-dimensional graph (2dg) format of later versions of Graph Tool. Its language is based on postscript and has advantages in that it can be quickly turned into a printable graph without the use of the Graph Tool drawing system. It can represent any type of graph. It is relatively user friendly, but this is hampered by the way the graph is represented. The file is basically a list of vertices followed by a list of edges. Every edge must involve a vertex that is listed. The edges become difficult to follow because of this. It however has trouble with colour and different style vertices cannot be represented. The file is better than daVinci because the file has better structured and can easily represent other types of graphs. Again the grammar for the file format along with an example file can be found in Appendix 2.

3.2.1.4 Graph Modelling Language (GML)

Michael Himsolt invented GML after discussions at the Graph Drawing conference in 1995 ([78] and [79]). It was proposed as the standard language for modelling graphs. It has since been accepted by a few packages; in particular the Library of Efficient Data Types [111], a library that contains implementations of various data structures such as graphs.

The language is simple to use and can be simply extended by adding new tags to it. Again it is basically a list of vertices and edges. The main advantage is that one file can contain several graph descriptions. Therefore it can be used to describe the common graph structures present in a graph and also the whole graph. However in order to perform this it will have to be modified by adding new tags and structures, and whilst still following the basic grammar it would not follow its original intended purpose and a GML file produced may not be loaded by other peoples interpreters. The basic grammar and example file is given in Appendix 2.

3.2.1.5 VCG Language

This language evolved from a project that tried to visualize compiler graphs ([146] and [147]). The tool that emerged from this was ahead of its time, its input language excellent. It allowed the user to specify which automatic graph layout algorithm to apply to the graph, and had a large set of attributes that applied to the graph as a whole or particular individual or groups of vertices and edges. Its only problem is that the language doesn't allow more than one graph to be specified; it would need extending to allow this. Within Sander [146] a grammar definition of the language is given, a version of which is given in Appendix 2.

3.2.1.6 General Comparison

When comparing the various file formats it is necessary to establish a list of features that is regarded as necessary in a file format, these features are listed in Table 10 along with a comparison of the file formats using these features.

The multiple graph feature is present if the file format allows several graphs to be described in one file format; it is if the file format allows graphs to be described that are not fully connected. This feature could be used to describe the common structures present in the graph. The best file format for doing this is GML because they can be clearly defined as separate structures. It is possible to do this in Graph Tool but they are not easily detectable from the main graph. It is desirable for the various graph properties to be described in the file format. For instance if it is possible to define which automatic graph layout algorithm is used to lay out the graph or subgraph. The other features are self-explanatory.

Feature	Adjacency List	daVinci	Graph Tool	GML	VCG
Graph Attributes					
Multiple Graphs	×	×	✓	✓	×
Directed/Undirected	×	×	✓	✓	×
Planar/nonplanar	×	×	×	✓	×
Layout Algorithm	×	×	×	×	✓
Title	×	×	×	✓	×
Unique label	×	×	×	✓	✓
Position	×	×	×	✓	✓
Size	×	×	×	×	✓
Vertex Attributes					
Vertex Colour	×	✓	✓	✓	✓
Text Colour	×	✓	✓	✓	✓
Text Style	×	✓	×	✓	×
Vertex icon	×	✓	×	×	×
Border Style	×	✓	×	×	×
Position	×	×	✓	✓	✓
Size	×	×	✓	×	✓
Label	✓	✓	✓	✓	✓
Unique Identifier	×	✓	✓	✓	✓
Edge Attributes					
Colour	×	✓	✓	✓	✓
Thickness	×	✓	×	✓	✓
Route	×	×	×	✓	×
Labels	×	✓	✓	✓	✓
Line Style	×	✓	✓	✓	✓
General					
Comments	×	×	×	✓	×
Character Set	**ISO**	**ISO**	**ISO**	**ISO**	**ISO**
Graph Tool Attached	×	✓	✓	×	✓
Allow General Graphs	✓	×	✓	✓	✓

Table 10 – A comparison of file formats

3.3 Graph Grammars

Graphs are one of the most important tools in computer science. They are useful because they are flexible, have a simple formal definition and have a natural visual representation that supports human cognitive capabilities. It is the formal recursive definition of graphs that allows graph grammars to be used. A graph is only useful if it is laid out 'nicely'. The aim of new tools for efficient constructions of 'nice' layouts of graphs stems from the desire to automate layout processes and to improve the quality of the layout. However many of the problems of automatic graph layout are NP Complete.

Most graph grammar parsers work in polynomial time and are well suited to outwit the NP completeness of optimal graph layouts. Graph specification languages have been written over the last decade. They often use graph grammars as their theoretical base.

This section provides a general introduction to the field of graph grammars and graph layout grammars. It provides a definition of them both, provides a general structure of a graph grammar, and shows how they are used.

3.3.1 Definitions

Brandenberg et al. [21] describes a graph grammar as: -

"A graph grammar consists of a finite set of productions of the form (A,R,C) where A is a vertex label, R is a finite graph and C is the connection relation. Vertex w with label A is replaced with graph R and C establishes edges between the neighbours of w and the vertices of R."

This was more formally written by Kaul [91] as: -

"Graph Grammar is a tuple $(\Sigma_V, \Sigma_T, \Sigma_E, P, S)$ *if*

1. $\Sigma_V, \Sigma_T, \Sigma_E$ *are finite alphabets for respectively vertices, terminal vertices and edge labels,* $\Sigma_T \subseteq \Sigma_V$
2. *P is a finite set of productions*
3. $S \ \varepsilon \Sigma V \setminus \Sigma T$ *is the start symbol"*

Graph grammars can be extended so that the graph can be laid out as well as described. The extension is called A Graph Layout Grammar which is a *"graph grammar but has a layout LS attached to it. LS contains many drawing specifications."* [21]. Brandenburg [22] in an earlier paper also provides a more formal definition as: -

"A Graph Layout Grammar consists of a polynomial graph grammar (grammar that runs in polynomial time) CG together with a layout specification LS. With each production P =(A, R, C), LS associates a finite set of layout constraints, $c_1, \ldots, c_q$. *A pair* (p,c_i) *is called a layout production. Each* c_i *defines a finite set of relations on the*

vertices of R, the left hand side A and the tuples from C that describe minimal distances between these objects in X- and Y- dimension. These relations must be consistent such that there exists a realisation in terms of a grid embedding of R, which can be extended to an embedding of (A, R, C). The distance constraints are additive, which means that $u.X \geq v.X+k$ $v.X \geq W.X+m$ *implies* $u.X \geq w.X +(k+m)$. *Here* $u.X$ *denotes the X-coordinate of the object u in some grid embedding. Moreover the constraints are complete i.e. each pair of vertices* $u,v \varepsilon V(R)$ *is related by at least one constraint."*

3.3.2 Example of Use

Graph Grammars can be used to represent a binary tree in the following manner

From a vertex there are two possible productions.

1) The terminal vertex (type a) is replaced with a tree with three vertices, one root vertex (type a) and two leaves (type A). The directed edges go from the root to the leaves. They are unlabelled. This entire structure is enclosed in a rectangle, which forms a vertex of type A. There is a single tuple (a,a) in the connection relation, which is shown in Figure 26a by the edge from the vertex label 'a' on top of the rectangle to the root.
2) Replace a vertex of type A with a terminal vertex type a, preserving the connections from type a. see Figure 26(b)

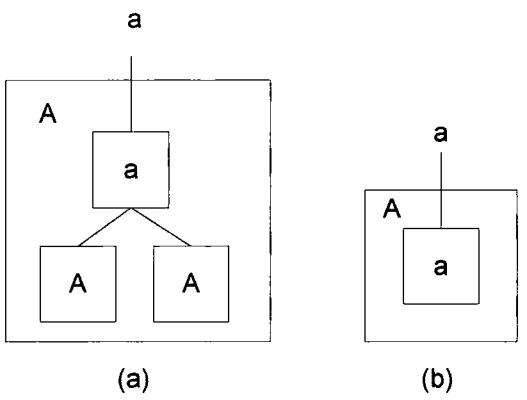


Figure 26 - The two production rules

From the above it can be seen that Graph grammars are a powerful tool. However it is questionable whether they are really useful for graph layout. They are only suitable for graphs that have recursive definitions, i.e. graphs that have a tree like structure. Brandenburg [21] suggests that they are therefore only useful for the following classes of graphs: -

- trees and the binary trees,
- series parallel graphs,
- partial k-trees for fixed k,
- maximal outer planar graphs,
- complete bipartite graphs,
- complete graphs, and
- flow graphs of programs.

They are not suitable for: -

- grids,
- planar graphs, or
- Directed a cyclic graphs.

They focus on the placement of the vertices; the routing of the edges is limited to straight lines. However there is no guarantee that such lines do not cross each other, although there may be extensions to allow this.

There are a few examples of the use of Graph Grammars in the real world. One that is quoted in Kahn [88] is that of a flowchart. Here a context free grammar for a flowchart developed by Lichtblau [104] is used to lay them out. It had limited success; the grammar is context free meaning that every case of flowchart cannot be generated.

3.4 Graph Drawing Tools

In the last fifteen years there has been an impressive growth in the number of automatic graph layout algorithms. There are many uses for these algorithms, one such use, and probably the main use is for displaying graphs either on a computer screen or on paper.

This is accomplished by the use of a graph tool. Generally tools fall into one of two types. A program that allows direct and interactive manipulation of such graphs on a high-resolution display, which is a graph editor and a subset of a graph editor that only allows the display of, and navigation in, a graph that is a graph browser.

This section provides a list of requirements for a typical graph tool, discusses interactive graph tools, and provides a classification of them and a summary of the tools found in the literature. It also gives the typical structure of a graph tool.

3.4.1 Graph Display Tools

There are many different uses for graph editor/display tools and a large number of papers describing them. There are many different features that can be provided by the graph tool, but a graph editor / browser must solve the following: -

- automatic layout of graphs,
- graph abstractions,
- adaptability, and
- persistence of graphs.

3.4.1.1 Automatic Layout of graphs

In Chapter 2 a summary of the area of automatic graph layout algorithms can be found. From this chapter it can be seen that automatic graph layout algorithms are a heavily researched area. It was also suggested that the main use of them was to display the graphs in a readable manner either on a display screen or on paper. Graph tools are where the majority of graph layout algorithms are used. It is probable that they were developed for this application.

3.4.1.2 Graph Abstractions

Even an adequate automatic graph layout algorithm may not be sufficient to make realistic graphs easy to read and manipulate. The sheer number of vertices, edges and

edge crossings may make graphical displays useless. Typically such graphs are black with edges. Magnification and scrolling may help, but additional facilities are needed to reduce the complexity of a graph. There are many such ways; the one that is described here came from Paulisch and Titchy [129] and is known as multi-level subgraph abstraction.

Multi-level subgraph abstraction is a method of grouping vertices and edges into subgraphs. The black box view displays the entire subgraph as a single vertex. This is useful when the subgraph is not relevant. The grey-box view is where the subgraph is visible but the edges from the outside go to the bounding box. In the white box view the subgraph is visible with all the connections left as normal, but the scroll bars allow only the movement around the subgraph.

3.4.1.3 Adaptability

Adaptability is the ability of the tool to be used to display other types of graphs e.g. pert charts and flowcharts. For each graph (diagram) there is a set of properties, for instance each could have different symbols and thickness of lines.

3.4.1.4 Persistence of Graphs

This is the ability of the tool to store the graphs (diagram) so that they can be viewed later. This is easy to implement and is present in all tools.

3.4.2 Interactive Graph Display Tools

Generally all modern graph tools allow the user to manipulate the graph in some way; in short they should be interactive. This is because modern computers: -

- can store a reasonable size of graph in the memory (less than 10000 vertices),
- have a powerful processor for processing them in some way, e.g. Graph Layout, and

- have increased ability to display graphs by using better displays. This is associated with the increased power of the processor to use new techniques to display them e.g. fisheye views.

There are many features that a tool should possess. Papakostas and Tollis [126] provide a study of these in orthogonal graphs; a summary is given here. Firstly the tool that supports interactive graph drawing should be able to create a drawing of a given graph under a given drawing standard. Secondly, the tool should give the user the ability to interact with the drawing in the following ways: -

- move a vertex around the drawing,
- move a block of vertices and edges around the drawing,
- insert an edge between two specified vertices,
- insert a vertex along with its incident edges, and
- delete edge, vertices or blocks.

The drawing of the graph at a given moment is called the current drawing, and the graph is called the current graph. The drawing resulting from the user change is called the new drawing, and the graph is called the new graph. Papakostas and Tollis [126] suggest there are various issues that should be taken into account before a new drawing is displayed, these are: -

- the amount of control the user has upon the position of a newly inserted vertex,
- the amount of control the user has on how a new edge will be routed in the current drawing connecting two vertices of the current graph, and
- how different the new drawing is when compared with the current drawing.

Baring this in mind Papakostas and Tollis [126] suggest the following four scenarios in interactive drawing: -

1) **full-control scenario** - the user has full control of the position of the new vertex in the current drawing. The user or the system can route the edges,

2) **draw-from-scratch scenario** - everytime a user requests to redraw the current graph, a completely new drawing is drawn redrawing the current graph using one of the popular drawing techniques,
3) **relative-coordinates scenario** - the general shape of the current drawing remains the same. The coordinates of some vertices and or edges may change because of the insertion of a new vertex, and
4) **no-change scenario** - in this approach the coordinates of the already embedded vertices, bends and edges do not change at all.

A more detailed description of these scenarios can be found in [126] and [127], they present algorithms for scenarios 3 and 4 applicable to orthogonal layouts. In Papakostas, Six and Tollis [125] an experimental comparison is made of scenarios 3 and 4 in terms of their performances.

3.4.2.1 Layout Stability

Automatic graph layout algorithms position vertices and edges and relieve the user of tedious and difficult manual layout. It tries to achieve a readable presentation. However Paulisch and Titchy [129] suggest that there are two problems with automatic layout, these are: -

- user preferences and application constraints are difficult to incorporate, and
- layout stability.

Most algorithms do not take the previous layout into account. After changing a graph, a new layout may be dramatically different to the previous one, causing loss of orientation. The four scenarios above play an important part in layout stability. Messinger [112] describes the difference between two graphs as how far the vertices have moved from their previous locations. Ideally when a user makes a change the automatic graph layout algorithm should just change the affected area, and therefore the difference between two layouts should be minimal. However, most change the whole graph. Incremental algorithms are a possible approach to take; these try to minimize the amount of recalculation needed to layout a graph. Incremental algorithms and minimising the difference between two layouts of graphs remain poorly explored. An

example of a tool that tries to achieve this is the EDGE system by Paulisch and Titchy [129].

3.4.3 Tool Classification

On surveying the literature, graph tools are divided into following six types: -

1. **layout algorithm** –an automatic graph layout algorithm,
2. **layout algorithm library** – a collection of automatic graph layout algorithms,
3. **graph library** – libraries of data structures that represent graphs,
4. **graph browser** –a way of displaying graphs on the screen,
5. **graph editor** –a way of displaying graphs and manipulating them, and
6. **graph layout system**- a program(s) that incorporates a layout algorithm library, graph library and a graph browser/editor

3.4.3.1 Graph Browser / Editor

In general a graph browser / editor can be represented by a figure similar to Figure 27. All systems take in a graph and output a graph. In some systems the graph processed by the automatic graph layout algorithm and then by the browser / editor which displays it to the user who can obtain a hard copy of it. In some systems a browser controls when to send it to the automatic graph layout algorithm. Graph layouts were obtained in batch before the increase in the power of computer systems in the 1980's. Graphs were sent to the automatic graph layout algorithm which would output another graph with the position of the vertices and edges stored within it or it would output in a file format that allowed it to be printed to a printer, e.g. postscript.

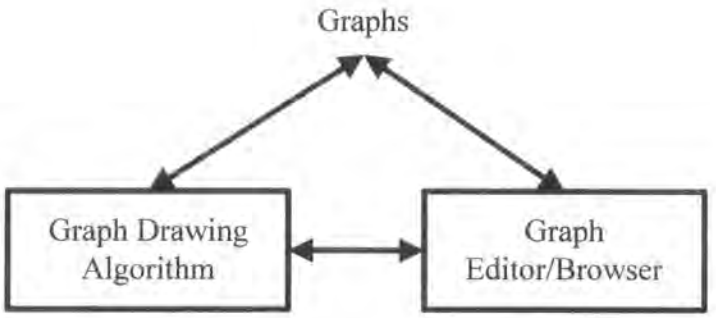


Figure 27 – A typical structure of a graph editor / browser

3.4.4 Graph Layout System

A typical graph layout system will be a variation of Figure 28. This system shows the best system is one that allows the user to implement many different algorithms, and draw many types of diagram, e.g. PERT Charts. A good example of such a system is ALF proposed by Bertolazzi et al. [12]. However many systems do not allow more than one type of diagram to be drawn. But they allow many algorithms to be applied to graph diagrams, such as GraphED or Graphlet [77] [80]. They have a system of representing algorithms and a limited store of them, but do not store different diagrams in the Diagram Model Store.

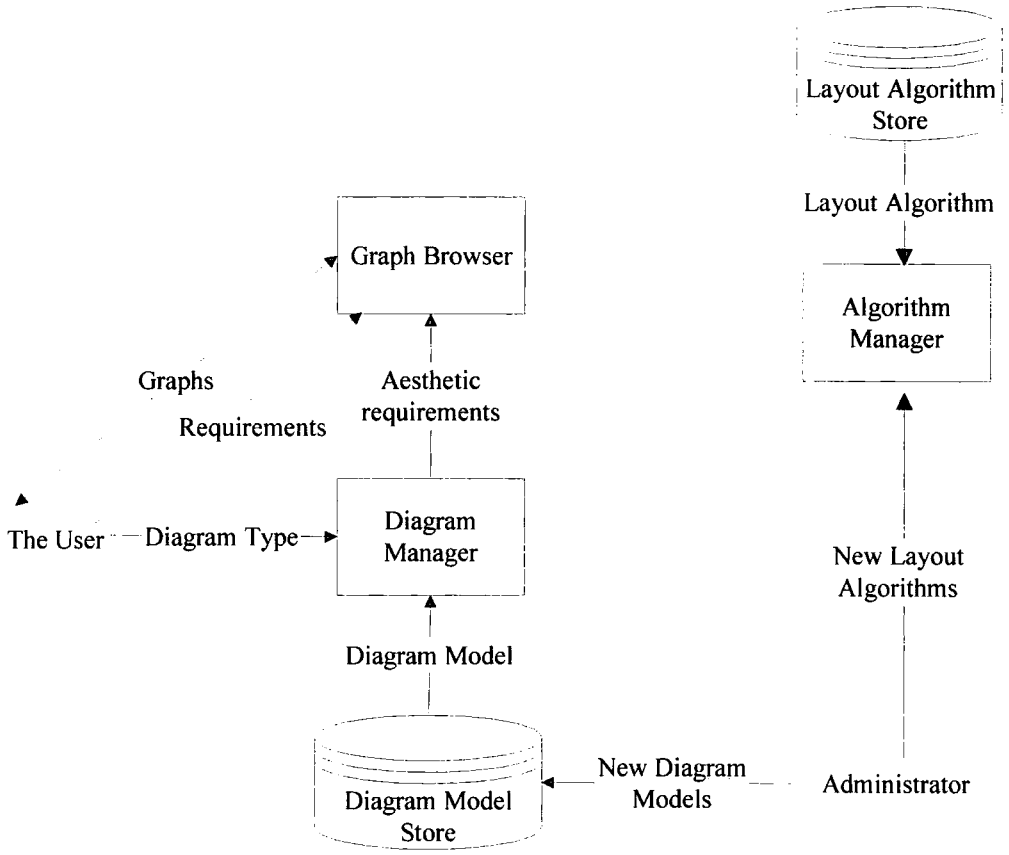


Figure 28 - A graph layout system modified from [12].

3.4.5 Tool Summary

DiBattista et al. [42] present many papers on tools. They are classified and a summary of the results is given in Table 11. Most computer science departments have a graph layout tool of some type as many systems have been developed.

Name	Paper	Algorithm	Input Class	Output Class	Classification
3Dcube	[128]	Independent	Any	3D	Graph Browser
ADOCS	[11]	Reingold And Tifford Tree and Eades Tree	Tree	Tree	Graph Browser
ALF	[12]	Independent	Any	Any	Graph Layout System
CABRI	[39]	None	Any	Any	Graph Editor
CABRI-Graph	[29]	Various (unlisted)	Any	Any	Graph Editor
CG	[110]	Grammar	N/A	N/A	Graph Browser
COMAIDE	[48]	Force Directed	Undirected	Undirected	Graph Browser
daVinci	[64]	Sugiyama	Any	Hierarchical	Graph Editor
D-ABDUCTOR	[158]	Sugiyama's Compound Graph	Compound	Compound	Graph Editor
DAG	[68]	Sugiyama	Any	Hierarchical	Graph browser
DynaDag	[123]	Sugiyama	Directed	Hierarchical	Graph Editor
EDGE	[129]	Independent	Any	Any	Graph Layout System
GD-Workbench	[28]	N/A	N/A	N/A	Graph Layout System
Giotto3d	[70]	Giotto – Tamassia	Directed Acyclic	3D Hierarchical	Graph Browser
Graph Editor Toolkit	[49]	N/A	N/A	N/A	Layout Algorithm Library
Graph Layout Toolkit	[108]	N/A	N/A	N/A	Graph Library
GraphED	[77]	Several (Unlisted)	Any	Any	Graph Layout System
Graphlet system	[80]	Any	Any	Any	Graph Drawing System
Grappa	[7]	Dot	Hierarchical	Hierarchical	Graph Browser
GROVE	[169]	Grammars	Any	Any	Graph Browser
Interactive Giotto	[24]	Giotto – Tamassia	Directed Acyclic	Orthogonal	Interactive Algorithm
LEDA	[111]	N/A	N/A	N/A	Graph Library
Link	[10]	Independent	Any	Any	Graph Layout System
Niche Works	[174]	Force Directed	Any	Circular, Hexagonal, Tree	Graph Editor
Optigraph	[83]	Force Directed	Undirected	Undirected	Graph Browser
SWAN	[179]	Bloesh (tree) Eades's Force Directed (undirected)	Undirected, Tree	Undirected Trees	Graph Editor
TOSCANA	[152]	None (unlisted)	Any	Any	Graph Browser
VCG	[146, 148]	Sugiyama Force Directed	Hierarchical	Hierarchical	Graph Browser

Table 11 - The classification of many layout systems

3.5 Graph Isomorphism

The ability to be able to compare graphs is important, e.g. robot vision and hand writing recognition. Graph isomorphism is a way of comparing graphs. The method of laying out graphs described in this thesis relies on the fact that graphs consist of common models. It is therefore necessary to search for these graphs so that they can be laid out using predefined layouts. Subgraph isomorphism techniques are methods of searching a graph for occurrences of a smaller graph, i.e. common models. Clearly subgraph isomorphism techniques use techniques developed in graph isomorphism. In this section the isomorphism problem is discussed giving a definition, a discussion of its complexity class, and the main algorithms used for performing isomorphism along with a comparison.

3.5.1 Definitions

3.5.1.1 Graph Isomorphism

There are several definitions of graph isomorphism. The definition to use depends on the type of graph. If the graph is defined as a set of vertices and edges, such as the definition given in Chapter 2 then graph isomorphism is defined by Gross and Yellen [73] as: -

"A graph isomorphism $f : G \rightarrow H$ *is a pair of bijections*

$$f_V : V_G \rightarrow V_H \text{ and } f_E : E_G \rightarrow E_H$$

Such that for every edge $e \in E_G$, *the function* f_V *maps the endpoints of* e *to the endpoints of the edge* $f_E(e)$.

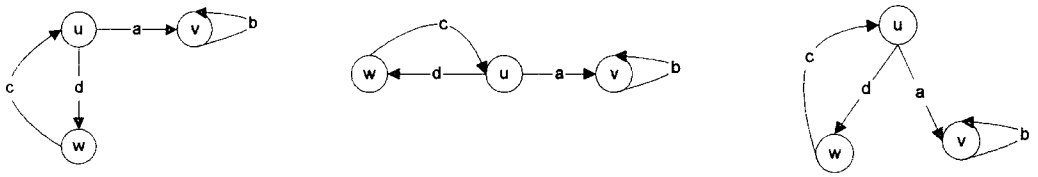
Two graphs G and H are said to be isomorphic if there exists an isomorphism from G to H. This relationship is often denoted by $G \cong H$ *or* $G \equiv H$*"*

Definition 1 -General graph isomorphism

For example given graphs G, H, I, J defined as: -

Graph $G=(V_G, E_G)$ where $V_G=\{u,v,w\}$ $E_G=\{(u,w),(w,u),(u,v),(v,v)\}$
Graph $H=(V_H, E_H)$ where $V_H=\{u,v,w\}$ $E_H=\{(u,w),(w,u),(u,v),(v,v)\}$
Graph $I=(V_I, E_I)$ where $V_I=\{u,v,w\}$ $E_I=\{(u,w),(w,u),(u,v),(v,v)\}$
Graph $J=(V_J, E_J)$ where $V_J=\{a,b,c\}$ $E_J=\{(a,c),(c,a),(a,b),(b,b)\}$

Graph G can be drawn so that it looks like Figure 29(a). Figure 29(b) shows a drawing of graph H and Figure 29(c) shows a drawing of Graph I. From the above definition they are all isomorphic u → u → u, v → v → v and w → w, edges a → a →a, b → b → b , c → c → c and d → d →d.



(a) Graph G (b) Graph H (c) Graph I

Figure 29 - A drawing of graph G, H and I

In graph J (shown in Figure 30) the labels are not the same as graphs G, however the vertex and edge structure are the same. In the above definition the graphs are isomorphic, however in many papers on the subject they are not. This is because in many definitions of a graph, functions are defined that assign labels to the edges and vertices. Therefore the labels are checked to be the same as well.

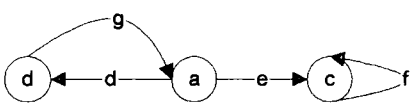


Figure 30 - Graph J

Vertex labelling helps with the efficiency of the isomorphism algorithms. Often when a graph is naturally unlabelled each vertex is given an individual label of its degree in an attempt to speed up detection of isomorphic graphs. In order to give all the necessary definitions of isomorphism a directed graph should be redefined so that a function is used to allocate labelling to the vertices and edges. The definition is taken from Bunke and Messmer [26] and is: -

A graph G is a four- tuple $G=(V,E,\mu,\nu)$, where

-V is the set of vertices

- $E \subseteq V \times V$
- $\mu : V \rightarrow L_V$
- $\nu : E \rightarrow L_E$

Definition 2 - A formal directed graph

Given this definition of a directed graph a bijective function $f : V \rightarrow V'$ is a graph isomorphism from a graph $G =(V,E,\mu,\nu)$ to a graph $G' =(V',E',\mu',\nu')$ if :

$$\mu(v) = \mu'(f(v)) \text{ for all } v \varepsilon V.$$

For any edge $e = (v_1, v_2) \varepsilon E$ there exists an edge $e' = (f(v_1), f(v_2)) \varepsilon E'$ such that $\nu(e) = \nu(e')$, and for any $e' = (v'_1, v'_2) \varepsilon E$ ' there exists an edge $e = (f(v'_1), f(v'_2)) \varepsilon E$ such that $\nu(e') = \nu(e)$

Definition 3 - Directed graph isomorphism

3.5.1.2 Subgraph Isomorphism

This definition is necessary because the ANHOF method does not need to compare one graph with another. It will need to search for all occurrences of a common model graph in the large graph. This is what is known as subgraph isomorphism. Subgraph isomorphism is a special case of graph isomorphism.

Informally a subgraph of a graph G is a graph H whose vertices and edges are all in G. Therefore in Definition 1 V_H and E_H are required to be subsets of V_G and E_G respectively H is therefore a subgraph of G when H is merely isomorphic to a subgraph of G. Formally using the terminology used in Bunke and Messmer [26]: -

Given a graph $G = (V,E,\mu,\nu)$, a subgraph of G is a graph $S = (V_s,E_s,\mu_s,\nu_s)$ such that : -

1. $V_s \subseteq V$
2. $E_s = E \cap (V_s \times V_s)$
3. μ_s and ν_s are the restrictions of μ and ν to V_s and E_s, respectively, i.e.

$$\mu_s(\upsilon) = \begin{cases} \mu(\upsilon) & \text{if } \upsilon \in V_s \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\upsilon_s(e) = \begin{cases} \upsilon(e) & \text{if } e \in E_s \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 4 - A subgraph

Using this definition subgraph isomorphism can now be defined. This is given in Definition 5.

An injective function $f : V \rightarrow V'$ is a subgraph isomorphism from a graph G to G' if there exists a subgraph $S \subseteq G'$ such that f is a graph isomorphism from G to S.

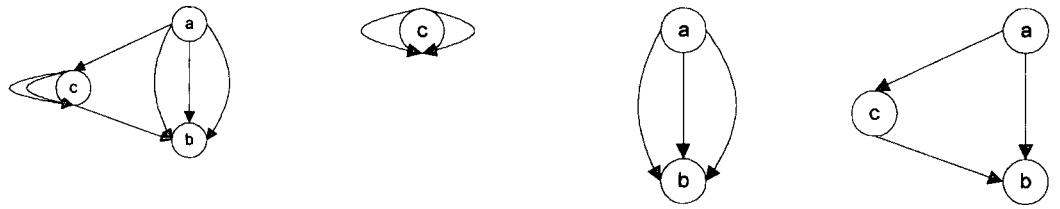
Definition 5 - Subgraph isomorphism

As examples consider the graph A defined as: -

V={a,b,c}

E={(a,c),(a,b),(c,b),(a,b),(c,c),(c,c)).

There are a number of ways of drawing graph A, one is given in Figure 31 together with some of the subgraphs of graph A. These are all isomorphic to a subgraph of graph A.



(a) Graph A (b) subgraph 1 (c) subgraph 2 (d) subgraph 3

Figure 31 - The graph A and its subgraphs

3.5.2 NP or P Complexity Class

Trying to decide whether such a pair of bijections (from Definition 1) exists is difficult. For certain pairs of simple n-vertex, q-edge graphs with n as small as six. A crude 'brute force' approach may require a combination of n! vertex bijections and q! edge bijections to see if one specification can be transformed into the other. This is why the graph isomorphism is regarded by most as being in the class NP [69]. However it is still an open question whether the graph isomorphism problem is in the complexity class P or NP. All algorithms that have been developed so far for the general graph isomorphism problem require, in the worse case, exponential time and are therefore in the class NP. Research in the last twenty years has shown that there are methods for graph isomorphism that behave reasonably well in terms of performance and become computationally interactable only in a few cases, therefore making the problem belong to class P. Further discussion on the topic is found in [98].

The graph isomorphism problem has been the focus of intensive research for three decades ([71] and [141]). There are basically two approaches that have been taken in order to find an efficient algorithm. The first approach is based on graph-theoretic concepts and aims at classifying the adjacency matrices of graphs into permutation groups. With this it is possible to prove that there exists a moderately exponential bond for the general graph [6]. Furthermore by imposing certain restriction on the graphs it is possible to derive algorithms that have a polynomial bound [81]. These methods, however, are not applicable in practice due to the large constant overhead of the adjacency matrix.

The second approach is more practical. It constructs the isomorphisms in a procedural manner. They are based on tree searching with backtracking, and work at worst in exponential time generally in polynomial time. It is these that are described in the next section.

3.5.3 Isomorphism Algorithms

Most practical graph isomorphism algorithms use standard tree searching techniques. One is based on a depth-first backtracking algorithm. It was first described by Corneil and Gotleib [36]. The method is as follows. Given two graphs G1 and G2 the vertices of G1 are mapped one after each other onto the vertices of G2 and after each mapping it is checked whether the edge structure of G1 is preserved in G2 by the mapping. If all vertices of G1 are successfully mapped onto vertices of G2 and G1 and G2 are equal size then a graph isomorphism is found. If G1 is smaller then G2 then a subgraph isomorphism from G1 to G2 is found. This performs well on small graphs but the number of steps required explodes exponentially.

3.5.3.1 Ullman's Algorithm

The most common graph isomorphism method used is that by Ullman [164]. It is an improvement on Corneil and Gotleib's [36] depth first search with backtracking. He combines the backtracking with a forward checking procedure. It reduces the steps required to search for an isomorphism and therefore increases its efficiency. In order to reduce the number of mappings that must be tested, it is better to start with a single vertex to vertex mapping and then gradually extend this mapping such that the resulting matching function always denotes a subgraph isomorphism. If it does not then the process backtracks to where it did and a further vertex is added. Further improvements were suggested by Ullman to increase its efficiency, known as forward checking. The basic idea of forward checking is to check for each mapping (v_i,w_{xi}) whether there exists at least one mapping for each future vertex v_j onto some vertex w_{xj} with $j>i$ such that the conditions for subgraph isomorphism hold true. The algorithm can be found in [164]. The advantage of this algorithm is that is simple to implement and relatively

efficient on both memory usage and running time. The speed is linearly dependent on the number of input graphs. It also performs well on unlabelled graphs.

3.5.3.2 Decision Tree

A decision tree is one of many new techniques that have been devised during the 1990's. It is suggested by Messmer in [114]. Given a set of model graphs, a decision tree is generated that represents, for each model graph, all possible permutations of the vertices. At run time, the isomorphisms or subgraph isomorphisms from an input graph to all model graphs can be found by a simple decision tree traversal. Checking to see if a given graph is isomorphic is then a simple task of searching the decision tree for the adjacency matrix of the graph. The algorithm for calculating the decision tree and checking if a graph is isomorphic is best described in Messmer and Bunke [115].

The advantages of the complete decision tree before pruning is that the performance is independent of the number and the connectivity of the model graphs, furthermore it is guaranteed to be only quadratic in the number of vertices of the input graphs. However the size of the decision tree that is compiled for the model graphs is exponential in the number of vertices of the model graphs, hence, only small model graphs (up to 19 vertices) can be handled. This fact has caused Messmer to suggest various techniques to prune the decision tree so that it is more manageable. The decision tree can be pruned depth and breadth wise, details can be found in [114].

Depth pruning is advantageous because the size of memory needed to store the decision tree is reduced, but because of the pruning the speed of the searching is increased. Breadth pruning cuts the size of the decision tree but it can only deal with small model graphs (up to 19 vertices). Pruning both depth and breadth wise increases efficiency of the algorithm and slightly larger graphs (up to 22 vertices) can be searched for.

3.6 Summary

In the chapter above some possible uses for graph theory and the automatic graph layout algorithms are suggested. It has presented graphs that are used as program comprehension tools, it has summarised the literature on graph specification languages, and categorised some tools. It has presented the main work on graph isomorphism, discussing the problem and giving two algorithms that solve it. In the next chapter these algorithms will be used to describe a new layout algorithm for call graphs, known as the ANHOF method.

4. The ANHOF Method of Call Graph Layout

4.1 Introduction

When comprehending programs, programmers make use of visualizations such as graphs, within these graphs they look for common structures. There are a number of graph layout algorithms incorporated into various graphs tools but none directly address the types of graphs in common use by software engineers. In particular they do not highlight the common structures found in call graphs. This chapter describes a method for automatic layout of large (greater than 150 vertices) call graphs, which conform to a set of aesthetics.

4.2 The Common Model Graphs

Fenton and Hill [56], amongst others, show that structural flowcharts consist of various common structures, such as those that map to sequence, decision and loop. A great deal of research has been carried out into call graphs in the University of Durham including their simplification [27]. Other research projects in Durham, such as AMES [19], have used call graphs as a program representation. Further research by Munro et al. [117] shows that there are common structures present in call graphs. Studying many graphs, they showed that call graphs consisted of at least five common structures. In this thesis the common structures have been given a layout. Therefore they have become what are referred to as common model graphs. The layout that has been assigned is a compromise between edge crossings and comprehension. The common model graphs are laid out so that there are minimum edge crossings and in a way that they can be understood and the common model graph stands out in the resulting graph.

There are two types of common model graphs, fixed and variable. Fixed common model graphs consist of a strict structure; they have a fixed number of vertices, edges and edge direction. Variable common model graphs consist of a variable number of vertices and

edges but have a common edge direction and structure. Figure 32 shows that there are nine common model graphs of which two are fixed and seven are variable. On inspection it becomes obvious that various common model graphs are made up of primitive common model graphs. These are common model graphs that cannot be simplified. There are five primitive common model graphs (Figure 32(a), (b), (c), (h) and (i)) of which two have a fixed structure and three have a variable structure. The primitive common model graphs are Triangle, Box, Fan In, Fan Out and Chain. However a Box consists of a Chain model and an extra vertex, the extra vertex making it difficult to simplify and is therefore a primitive common model graph.

These common model graphs are discussed in the section below. It should be noted that the definitions of the common model graphs below are trying to be as restrictive as possible. The theory behind this layout method is that a graph should be made up of as many common model graphs as possible. Therefore a common model graph should minimize the number of vertices that it contains, in order for the automatic graph layout algorithms to work at their peak performance and to aid understanding of the graph. In the diagrams below the primitive common model graphs are coloured differently so that they can be identified in future example graphs. If in a diagram a dashed line is present this means that the edge is not necessary. For each model graph an example interpretation is given that relates the graph to the structure core.

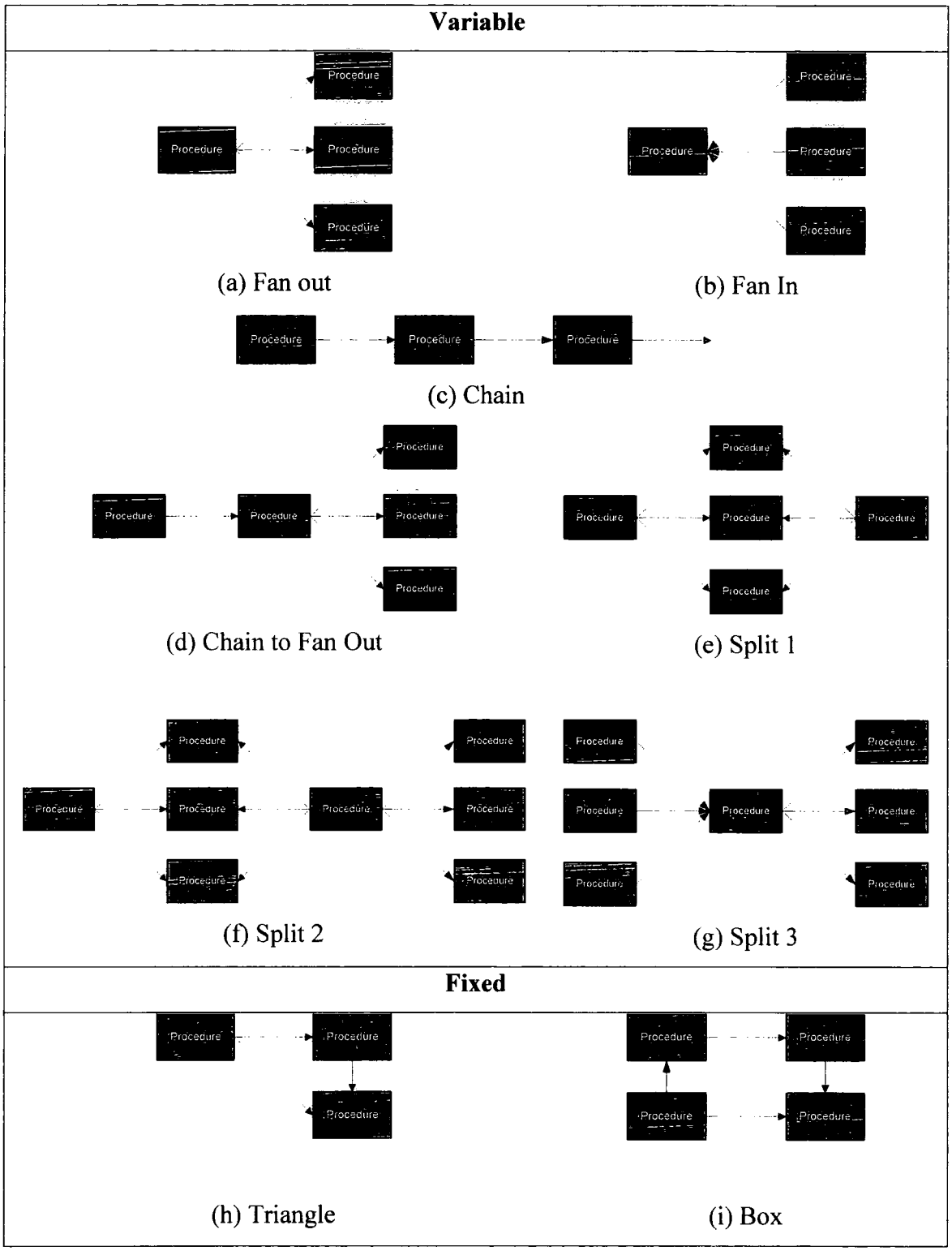


Figure 32 - The common model graphs present in a call graph

4.2.1 Variable Common Model Graphs

These are common model graphs that have a fixed edge direction but can involve any number of vertices. They all rely on two properties of a vertex described by Henry and Kafura [76]. The number of edges that lead from a given vertex, known as the fan out number of a vertex, and the number of edges that come into a vertex, known as the fan in number of the vertex. Using this information about a vertex, various common model graphs become obvious. The seven common model graphs, three of which are primitive common model graphs (Figure 32(a), (b), and (c)), are discussed below. In describing each of the model graphs a set of numerical values can be defined to parameterise each graph. These values will be detailed by experimentation later in this thesis in Chapter 7. The parameters are given below.

- **Fanoutlevel** – Measure of the fan out value of a vertex. It should be a value greater or equal to two.
- **Faninlevel** – Measure of the fan out value of a vertex. It should be a value greater or equal to two.
- **Chainlevel** – The number of vertices that flow in a chain like manner. It should be a value greater or equal to two.
- **Chainfanoutlevel** – The number of vertices that a vertex should fan out to at the end of a Chain. It should be a value greater or equal to two.
- **Lengthofchain** – The number of vertices that flow in a chain like manner. It should be a value greater or equal to two.
- **Commonfanoutnumber** – This is the number of vertices that the two main vertices commonly fan out to in a Split 1 model. It should be a value greater or equal to two.
- **Commonsplit2fanoutlevel** – This is the number of vertices that the two main vertices commonly fan out to in a Split 2 model. It should be a value greater or equal to two.
- **Split2fanoutlevel** – This is the number that one of the vertices fan out to in a split 2 model. It should be a value greater or equal to two.
- **Split3fanoutlevel** – This is the number of vertices that fan out from the main vertex in a Split 3 model. It should be a value greater or equal to two.
- **Split3faninlevel** – This is the number of vertices that fan into the main vertex in a Split 3 model. It should be a value greater or equal to two.

4.2.1.1 Fan Out Common Model Graph

4.2.1.1.1 Description

This common model graph is the most common in a call graph because of its hierarchical structure. A Fan Out common model graph consists of one main vertex that flows to a number of vertices. There will be a lower limit to this number of vertices and will be defined using the *'Fanoutlevel'* parameter above to avoid conflict problems of detecting other common model graphs.

4.2.1.1.2 Layout

The common model graph is laid out in the manner given in Figure 33 with the main vertex centred above the ones it flows out to. In Figure 33 the father vertex is labelled 'Procedure' and the children vertices are labelled 'Procedure 1', 'Procedure 2', and 'Procedure 3' to 'Procedure N'. The model is coloured dark green to aid future identification.

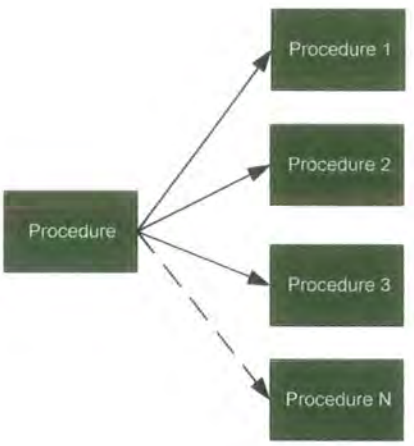


Figure 33 - A Fan Out common model graph

4.2.1.1.3 Example

A C program consists of a function that is always present, known as the 'main function'. In large computer programs this function will call other functions giving the hierarchical structure common in call graphs. It is this information that is represented by a Fan Out common model graph. The functions that are immediately called inside the main

function are represented by the vertices that the main vertex (vertex labelled 'Procedure') flows to (vertices labelled 'Procedure 1', 'Procedure 2', 'Procedure 3' and 'Procedure N' in Figure 33).

4.2.1.2 Fan In Common Model Graph

4.2.1.2.1 Description

This common model graph is another model that is implied in the hierarchical nature of a call graph. A Fan In common model graph consists of a set number of vertices (known as the fan in vertices) flowing into one main vertex. Again there will be a lower limit to the number so that the model graph will be detected reliably; this limit will be set by the parameter '*Faninlevel*' defined above.

4.2.1.2.2 Layout

The common model graph is laid out in the manner given in Figure 34, with the main vertex centred below the fan in vertices, where the flow goes across the plane. The main vertex is labelled in Figure 34 as 'Procedure' and the fan in vertices are labelled 'Procedure 1', 'Procedure 2', and 'Procedure 3' to 'Procedure N'. The vertices are coloured yellow to aid future identification.

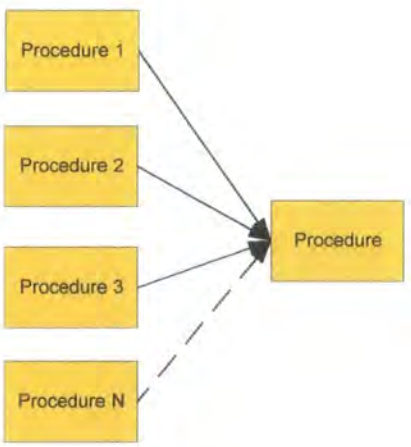


Figure 34 - A Fan In common model graph

4.2.1.2.3 Example

A Fan In common model graph is commonly present if a call graph is not simplified by removing the standard library functions. For instance if the C function 'printf' is present, many functions will call on that function. In a call graph the functions that call 'printf' will be the fan in vertices (vertices labelled 'Procedure 1', 'Procedure 2', 'Procedure 3' and 'Procedure N' in Figure 34) and 'printf' will be the main vertex (vertex labelled 'Procedure') in a Fan In common model graph. It is also indicative of a commonly used function that may for example report an error message.

4.2.1.3 Chain Common Model Graph

4.2.1.3.1 Description

A Chain common model graph implies the simplest structure in a program, one procedure invoking another and that invoking another etc, it is therefore potentially common in call graphs. A Chain common model graph consists of a series of vertices that have a fan out value of exactly one. The Chain common model graph will end with a vertex that has a fan out value not equal to one. Ideally the start vertex should have a fan in value of zero. Each vertex can have a fan in value of any value; this is because if it is restricted to zero then a Chain model is rarely found in call graphs. This causes a vertex to flow to the next one and that one to flow to the next etc therefore resembling a chain. The length of the chain is the minimum number of vertices that are involved. It is set and defined under the parameter '*Chainlength*'.

4.2.1.3.2 Layout

The vertices of a Chain common model graph should be aligned so that the top edges are aligned, with equal spacing in between the vertices. An example Chain common model graph layout is shown in Figure 35, it is again coloured light green to aid future identification.

Figure 35 - A Chain common model graph

4.2.1.3.3 Example

A Chain common model graph is common in data processing applications when the main function calls another function to do a small chunk of data processing which in turn calls another function that performs some sort of output / input. On detection of a Chain model a question may be asked about the programmer, does the programmer understand hierarchical structures that indicative of good programming structure?

4.2.1.4 Chain to Fan Out Common Model Graph

4.2.1.4.1 Description

This consists of a combination of a Chain common model graph and a Fan Out common model graph. It is merely a redefinition of the Chain common model graph, in that less strict rules are placed on the end vertex of the chain. The Chain common model graph consists of a series of vertices that have a fan out value of exactly one, the end vertex has a fan out value greater or equal to a number. The end vertex and the vertices that it flows out to form a Fan Out common model graph. Again there will be a limit to this number and the limit is set under the parameter *'Chainfanoutlevel'*. The length of the chain is the minimum number of vertices that is involved in the chain. The lower limit of this number is set by the parameter *'Lengthofchain'*. It has the same properties as 'Chainlength' but is given a separate name so the settings can be unique.

4.2.1.4.2 Layout

The top of the vertices associated with the Chain model (vertices labelled 'Procedure', 'Procedure 1' and 'Procedure N' in Figure 36) should be aligned horizontally; they should be position so that they are centred over the children of the Fan Out common model graph (vertices labelled 'Procedure N+1', 'Procedure N+2' and 'Procedure N+M' in the diagram). An example layout of the common model graph is given in

Figure 36. The composition of the common model graphs are highlighted by the vertex colour, the Chain model being coloured light green and the Fan Out model being coloured dark green. The vertex labelled 'Procedure N' is a member of both the Chain common model graph and Fan Out common model graph

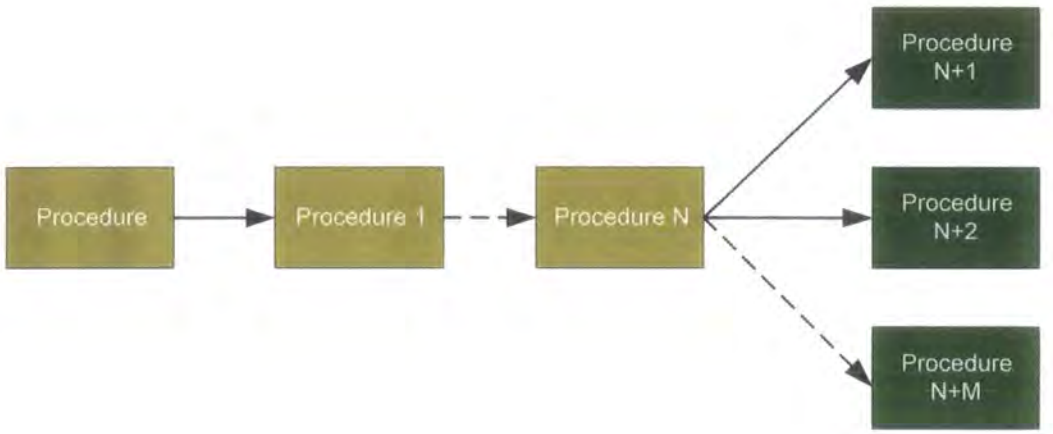


Figure 36- A Chain to Fan Out common Model Graph

4.2.1.4.3 Example

A Chain to Fan Out common model graph is common in data processing applications when the main function calls another function to perform some data processing (the Chain common model graph labelled 'Procedure', 'Procedure 1'and 'Procedure N' in Figure 36). This step is then broken down into smaller steps, each step requiring another function (the Fan Out common model graph labelled 'Procedure N+1', 'Procedure N+2' and 'Procedure N+M'), each function is represented as a vertex that flows from the step vertex (labelled 'Procedure N').

4.2.1.5 Split 1 Common Model Graph

4.2.1.5.1 Description

A Split 1 common model graph is a combination of two Fan Out Models. It is where two main vertices flow to the same set of vertices. The number of common vertices has to be greater or equal to a number. The lower limit to the number is set by the parameter *'Commonfanoutnumber'*.

4.2.1.5.2 Layout

An example layout of a Split 1 common model graph is shown in Figure 37. The common vertices are labelled as 'Procedure 2', 'Procedure 3' and 'Procedure N'. The vertices should be laid out so that the main vertices (labelled 'Procedure 1' and 'Procedure N+1' in the diagram) are laid out centrally either side of the common vertices.

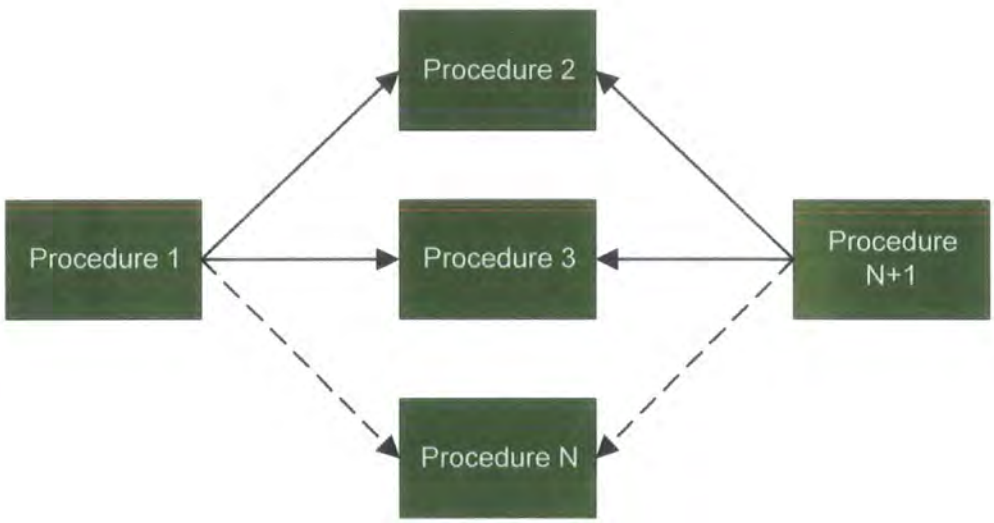


Figure 37 - A Split 1 common model graph

4.2.1.5.3 Example

This structure is common where two procedures call a similar set of functions to perform similar tasks. Sometimes it is an indicator that the procedures do the same thing. For example if a procedures does a certain task on a data type that requires many procedures to perform and another procedure does the same processing on a different data type using the same procedures. Another example of the use of a Split 1 model is if vertices labelled 'Procedure 2', 'Procedure 3' and 'Procedure N' manipulate an equation and vertices labelled 'Procedure 1' and 'Procedure N+1' set up values for it.

4.2.1.6 Split 2 Common Model Graph

4.2.1.6.1 Description

This is similar to a Split 1 common model graph in that two main vertices flow out to common vertices but one vertex also fans out to more, so the common vertices are a subset of the set of vertices it fans out to. The common number of vertices has to be greater or equal to a number, set by the parameter '*Commonsplit2fanoutlevel*'. The father vertex that fans out to more vertices should have a fan out value greater or equal to a number, set by the parameter '*Split2fanoutlevel*'.

4.2.1.6.2 Layout

An example layout of a Split 2 common model graph is shown in Figure 38. In this diagram the common vertices are labelled 'Procedure 2', 'Procedure 3', and 'Procedure N'. The main vertices being labelled 'Procedure 1' and 'Procedure N+1'. The extra vertices flow from the vertex labelled 'Procedure N+1' and are labelled 'Procedure N+2', 'Procedure N+3' and 'Procedure N+M'. It should be laid out so that the father vertices are centred over the common vertices and the extra vertices.

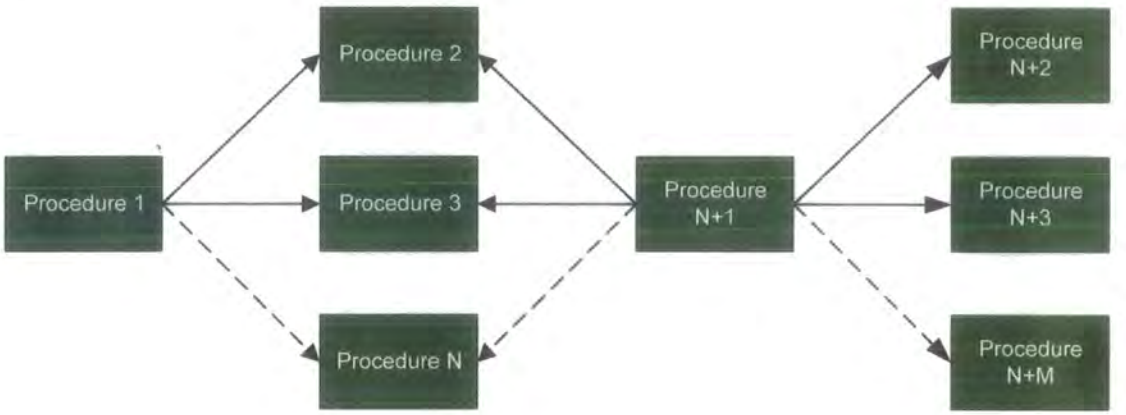


Figure 38 - A Split 2 common model graph

4.2.1.6.3 Example

It is again common where two procedures call a similar set of functions to perform similar tasks. For example if a procedures does a certain task on a data type that requires many procedures to perform and another procedure does exactly the same processing on a different data type using the same procedures as the other procedure but requires more procedures to do the processing.

4.2.1.7 Split 3 Common Model Graph

4.2.1.7.1 Description

A Split 3 common model graph is simply a vertex that has a number of vertices both coming in and out of it. The lower limits to these numbers are set by the parameters '*Split3fanoutlevel*' and '*Split3faninlevel*' respectively. It therefore consists of a Fan In common model graph and a Fan Out common model graph with common father vertices.

4.2.1.7.2 Layout

The common model graph should be laid out so that the main vertex is in the centre and it is centred over its children. The layout can be found in Figure 39. In this figure the common model graphs are coloured in their respective colours to aid identification. The father vertex is labelled 'Procedure N+1' and coloured black because it is a member of both common model graphs. The fan in vertices are labelled 'Procedure 1', 'Procedure 2' and 'Procedure N', and the fan out vertices are labelled 'Procedure N+2', ' Procedure N+3', and 'Procedure N+M'.

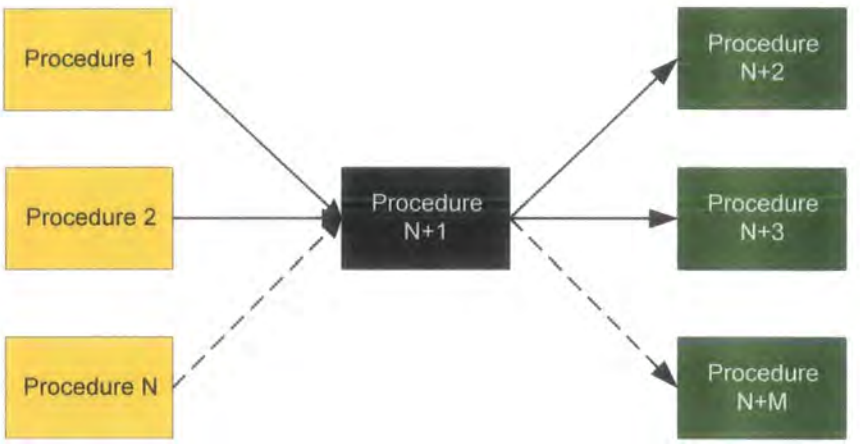


Figure 39 - A Split 3 common model graph

4.2.1.7.3 Example

This is present where a procedure relies on the input of two or more procedures to perform its task and then other procedures need the information that will be provided by calling that procedure. So, it may be a type operation that processes a file where each part of the file is processed by a procedure and the main procedure combines the results of these procedures together. This type of operation is required by many other procedures in the program.

4.2.2 Fixed Common Model Graphs

A fixed common model graph is a model that has a fixed structure; they have a fixed number of vertices, edges and edge direction. In the section below two examples of fixed common model graphs will be discussed, a Triangle and Box common model graph.

4.2.2.1 Triangle Common Model Graph

4.2.2.1.1 Description

A Triangle common model graph consists of three vertices. One main vertex that flows to the other connected vertices. It is commonly found in the midst of a Fan Out common model graph. It is a primitive common model graph because it cannot be simplified and involves a fixed number of vertices.

4.2.2.1.2 Layout

It should be laid out so that the top of the main and one of the other vertices are horizontally aligned. It differs from a normal layout because the father vertex is not centred over its children; this is because it is easier to impose a fixed layout to fixed common model graphs. In this common model graph one vertex is called by both the other two vertices meaning that it is likely to be a small function, and the other two procedures will be using it to get a small result. A programmer will read graphs firstly left to right on the page and then top to bottom. Therefore it is necessary that the important vertices are place nearest the left hand edge of the page then nearest to top edge of the plane, hence the layout of the common model graph. The layout is given in Figure 40.

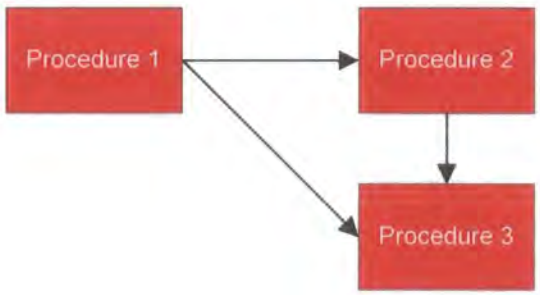


Figure 40 - A Triangle common model graph

4.2.2.1.3 Example

A Triangle common model graph is common where a procedure needs two sub-procedures and one relies on the other for results, for instance in a mathematical equation.

4.2.2.2 Box Common Model Graph

4.2.2.2.1 Description

There are 16 combinations of four unlabelled vertices. Two of these combinations are cycles and are not allowed by the subgraph isomorphism algorithms. Nine combinations are simple isomorphisms of each other, leaving the five combinations in Figure 41. When searching call graphs it was found that the combinations in Figure 41(c)(d)(e) were not found in call graphs and therefore did not feature in the library of common model graphs. Figure 41(a) is a Box common model graph discussed below. Figure 41(b) is a Split 1 common model graph but the settings of the parameters may not detect it as such, again it was not found in call graphs.

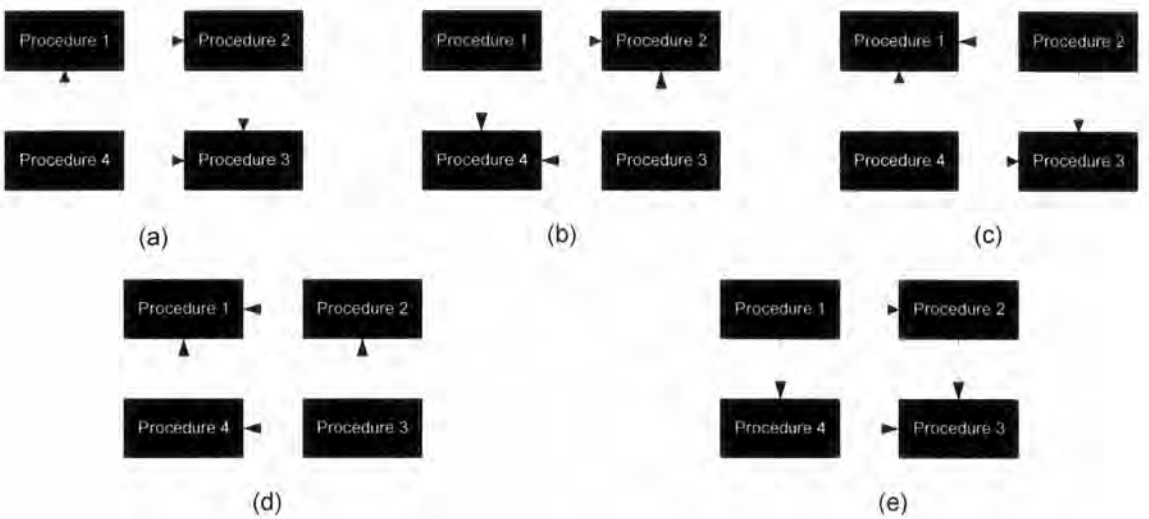


Figure 41 - The independent variations of four vertices

A Box common model graph consists of four vertices and not a cycle. The first vertex (labelled 'Procedure 1') is connected to the second vertex (labelled 'Procedure 2'), which is connected to the third vertex (labelled 'Procedure3'). The fourth vertex (labelled 'Procedure 4') is connected to the first and third, therefore it is acyclic.

4.2.2.2.2 Layout

A Box model should be laid out so that vertices 1 and 4 are vertically aligned, as are 2 and 3. Vertices 1 and 2 should be horizontally aligned, as should 4 and 3. If the 'Chainlength' variable is set to 2 or 3 a chain is present (vertices labelled 'Procedure 1', 'Procedure 2' and 'Procedure 3'. However because of its structure the common model graph is present in this section, and is marked as a primitive common model graph because it involves another vertex that will not be part of another common model graph. An example layout is given in Figure 42.

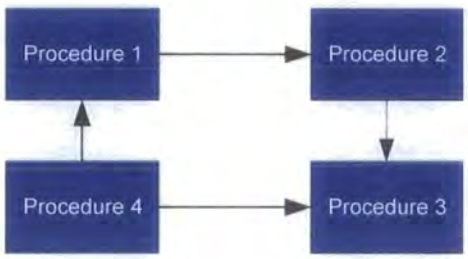


Figure 42 - A Box common model graph

4.2.2.2.3 Example

A Box common model graph is common in data processing applications when a start procedure calls another procedure to perform some data processing which in turn calls another function to performs output / input. Another procedure calls that output procedure and the start procedure

4.3 The ANHOF Method

The ANHOF method brings together areas of graph layout to highlight common structures in calls graphs. The main aims of the ANHOF method are to improve the understanding of the graph and its metrics. The improvement in understanding of call graphs is gained by making the common structures obvious in its layout. Reduction in the number of edge crossings is achieved by breaking down the graph into smaller units

where automatic graph layout algorithms work more effectively. The common structures have an associated 'good' layout. The good layout is a compromise between low edge crossings and easy comprehension. These techniques are commonly called a divide and conquer approach and were first described by Messinger et al. [113]. It is where first the graph is divided into subgraphs; the graphs are then laid out and then recombined into a new and improved graph. The method presents two problems. Firstly how to partition the input graph into subgraphs and, secondly, how to layout the subgraphs.

To partition the graph Messinger et al. [113] suggested two methods. 1) Application specific partitions 2) graph theoretic partitions such as those discussed by [124]. In the ANHOF method an application specific partition is used in such a way the graph is partitioned in to a set of common model graphs present only in that graph type.

Automatic graph layout algorithms can be classed into declarative approach, algorithmic approach, or a combination of the two. An algorithmic approach is where a strict algorithm is applied to the graph. This approach has generally produced some good layouts for specific classes of graphs. They however rarely work outside of their intended class. Layout requirements are generally hard coded into them making them difficult to customise. They rely on the structure of the graph. A declarative approach is where the rules and constraints to be placed on the graph are clearly specified. They are then applied to the graph. Constraints however often apply to a graph in many unforeseen ways and they tend to be inefficient. Paulisch and Titchy [129] suggest that it is difficult to incorporate users and application constraints into one automatic graph layout algorithm. An integrated approach is a combination of both of the approaches. It is where the constraints are defined in terms of the options of algorithmic approaches' algorithm.

The ANHOF method presents a means of automatically laying out the hierarchical / tree like structures of call graphs, meaning that an algorithmic approach could be employed. In order to make the ANHOF method as general as possible, however, an integrated approach is employed, where constraints and aesthetics such as vertex spacing and the layout options of the automatic graph layout algorithms are specified. It employs a

straight-line standard from edge routing and intends to apply the following aesthetic to the layouts: -

- minimize the crossings of edges,
- the flow of the graph will go from left to right,
- the area of the graph will be minimized,
- vertices on the same level will be placed on the same horizontal line,
- fathers will be centralised above their sons.

The divide and conquer method of call graph layout uses the ANHOF method. This is a four-part process that is shown in Figure 43. The common model graph detection that is performed by the Graph Isomorphism System. This produces a list of all the matches to the models. The Match Analyser removes any invalid ones, filtering the matches to the common model graphs. This creates a graph representation and a list of valid models. The list of valid matches is an unused output available for use in other tools. The graph representation contains all the valid matches and details of the whole graph. It is this representation that is used in laying out the graph with the Graph Layout System and is then displayed with the Graph Display System. Each section of the process will be expanded in the sections below.

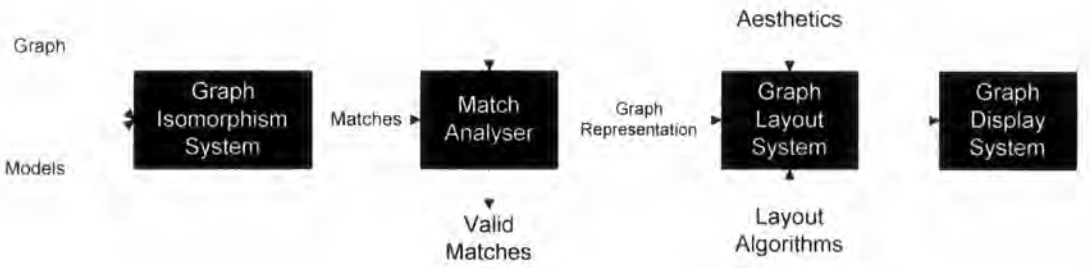


Figure 43 – The ANHOF method of call graph layout

4.3.1 Graph Isomorphism System

There are two types of common model graphs found in call graphs, fixed and variable. In order to detect the different common model graphs type two sorts of detection methods are needed. The techniques of subgraph isomorphism described in Chapter 3

should be employed to extend the library of fixed common model graphs. The algorithms by Ullman [164] and Messmer [114] are used and search for smaller adjacency matrices in the larger one representing the graph. These work well with the size of common model graph (less than 20 vertices) suggested. It is therefore only a matter of calculating its adjacency matrix to detect any fixed common model graph and using either algorithm and to find out whether it is present in the call graph.

The detection of the variable common model graphs relies on processing the sets of edges and vertices. The processing of the sets is a two-step process. First, calculate the fan in and fan out values of each vertex, and then process this information to detect the various common model graphs according to their definition. The common model graph definitions should be written in a simple language to allow future expansion of the common model graph library that is then processed. The fixed common model graphs could also be detected using the system but are best detected using the subgraph isomorphism algorithms because some common model graphs cannot be expressed in terms of the fan out and fan in values of its vertices. For instance a Box common model graph is best described in terms of its adjacency matrix because two vertices have the same fan in and fan out information.

Detecting the common model graphs in a graph can be divided further into a three-step process. The process is given in the algorithm **SearchForModelGraph** (Algorithm 2).

SearchForModelGraph(*G, FM, VM*)

Where G=(V,E), FM = FixedModels and VM = VariableModels

1) For each fixed common model graph *FM*: -
   a) Form adjacency matrix of *FM*
   b) Search Graph *G* using a subgraph isomorphism algorithm for *FM*
   c) Output the common model graph matches found to list *Matches*
2) Calculate the fan in and fan out information for each vertex in *G*
3) For each variable common model graph *VM*: -
   a) Take common model graph definition of *VM* and: -
   b) Search fan in and fan out information for correct vertices
   c) If the definition requires the edge information then check the vertices found in step b) for the correct edge information

d) Output the common model graph matches found to list *Matches*

4) Return *Matches*

Algorithm 2 – The process of searching for the common model graphs

4.3.2 Match Analyser

The Graph Isomorphism System will find a large number of possible matches to the common model graphs. If there are too many matches then the layout process will be hampered. If there are too few, the common model graphs searched for by people comprehending programs will not become obvious, therefore not improving the graph layout from the program comprehension point of view. The number of vertices being passed to the automatic graph layout algorithm could be too large for the algorithms to work successfully. Therefore the layout of the graph in terms of its aesthetics will not be improved, and it is necessary to define what a valid match is. There are many definitions of a valid match, for instance a common model graph cannot involve more than a set number of vertices. One of the problems of using a divide and conquer method of graph layout is how to cope with a vertex being a member of two or more common model graphs. In order to overcome this problem a valid match is a common model graph that does not include a vertex that is a member of another common model graph. The Match Analyser therefore processes models so that they do not involve the same vertices. **AnalysisOfMatches** (Algorithm 3) performs this analysis on a first come, first served basis. The effectiveness of the algorithms is susceptible to the order in which the common model graphs are processed; this order is investigated in future sections. In **AnalysisOfMatches** (Algorithm 3) the valid matches are an output in *ValidMatches* and *Representation.*

AnalysisOfMatches(*AllMatches, G*)

Where *AllMatches* = *Matches* and *G*=(V,E)

1) Let *AllVertices* = V
2) For each match in *AllMatches:* -
    a. Get the list of vertices *InvolvedVertices* involved in the match

b. For each vertex *InvolvedVertex* of *InvolvedVertices*: -
    i. If *InvolvedVertex* is a member of *AllVertices* then
        AllVertices = *AllVertices* \ *InvolvedVertex*
    ii. Else Current match is invalid
c. If no more vertices in *InvolvedVertices* then current match is valid
d. Add to *ValidMatches*
e. Add to *Representation*

3) Return *ValidMatches*
4) Return *Representation*

Algorithm 3 - The process of filtering of matches

A representation of the graph is the output from the Match Analyser. It requires very little processing and is necessary so that the output from Match Analyser is stored and can be viewed and altered before it is laid out by the Graph Layout System. The representation should allow the graph to be described in terms of its common model graphs.

The representation forms a simple domain language for describing a call graph. The design of the representation used will follow the constituents of a good language discussed in Pratt and Zelkowitz [132] and Bentley [9] mainly simplicity, orthogonality, and naturalness. The representation must be simple to understand to a user in the field of graph theory by making any terms used in the representation the same as any used in the field, and making a construction applicable to many areas. The key information that is to be represented is the vertices and edges of the graph, the common model graphs in terms of its name, algorithm to be used and vertices involved. Details of the representation will be discussed in the next chapter.

4.3.3 Graph Layout System

The Graph Layout System involves processing the representation of the graph that includes vertices and edges that are in common model graphs and those that are not in any model. The common model graphs are laid out using their associated algorithms and the rest of the graph is laid out using either a standard automatic graph layout

algorithm or a custom one for hierarchical structures. However the common model graphs should be taken into account when laying the rest out.

This is achieved by forming a new graph. Each common model graph is laid out using its associated automatic graph layout algorithm. All the vertices of the model are collapsed into one vertex in the new graph, all edges that went to/from vertices in the common model graph from/to vertices outside the common model graph now go to/from the one vertex. The height and width of the common model graph is calculated after it is laid out and the vertex that represents the common model graph in the graph is given the same properties. All the vertices and edges that are not part of the common model graphs in the original graph are added to the new graph as well. The whole graph is now laid out using a standard or customised automatic graph layout algorithm. The graph is now rebuilt, taking each vertex representing a common model graph, the vertex is deleted and the vertices and edges in that common model graph are put back in its place. Each vertex in the model is offset by the position in the horizontal and vertical axis of the representing vertex. Therefore the layout of the model is maintained, each member vertex is placed in a position relative to its representative vertex, so maintaining the layout of the whole graph. The process is given in **LayoutRepresentation** (Algorithm 4).

The automatic graph layout algorithms in the ANHOF method employ a straight line standard of graph layout. This means that edges connect two vertices by the shortest path, it is irrelevant whether it crosses another edge or vertex. None of the algorithms in the section below specifically route edges. There is no step in **LayoutRepresentation** (Algorithm 4) that routes the edges, unless the standard automatic graph layout algorithm in step 8 or a model's automatic graph layout algorithm in step 5d performs it.

LayoutRepresentation(*R*)
Where R = Representation

1. Get the whole graph *G*(*Vertices*,*Edges*) from *R*.
2. Let *Unusededges* = *Edges*.
3. Let *Unnusedvertices* = *Vertices*.
4. Let graph D (V_2, E_2) be a new graph.

5. For each common model graph M in R: -
    a. Get vertices (V_1) and edges (E_1) that are involved in the common model graph from R
    b. Get the name N of the common model graph M from R.
    c. Get the automatic graph layout algorithm to use on M from R.
    d. Layout graph (V_1, E_1) using the common model graph's automatic graph layout algorithm.
    e. Calculate the height H and width W of M.
    f. Add one vertex *Mvertex* to V_2, labelled N, with the height = H and width = W.
    g. Change all edges that go to / from a vertex involved in M to a vertex outside, so that they go to / from *Mvertex*.
    h. Set *Unusededges* = *Unusededges* \ E_1.
    i. Set *Unusedvertices* = *Unusedvertices* \ V_1.
6. Let $V_2 = V_2 +$ *Unusedvertices*.
7. Let $E_2 = E_2 +$ *Unusededges*.
8. Layout graph $D(V_2,E_2)$ using a standard automatic graph layout algorithm.
9. For each vertex $vertex_1$ in $D(V_2,E_2)$ that represents a common model graph : -
    a. Set *Xpos* = X position of $vertex_1$.
    b. Set *Ypos* = Y position of $vertex_1$.
    c. For each vertex *vertex2* in the common model graph: -
        i. Let $V_2 = V_2 + vertex_2$.
        ii. Let X position of $vertex_2$ = X position of $vertex_2$ + *Xpos*.
        iii. Let Y position of $vertex_2$ = Y position of $vertex_2$ + *Ypos*.
    d. $V_2 = V_2 \setminus vertex1$.
10. Change all edges in graph $D(V_2,E_2)$ so that they go to/from all of the original vertices in graph G, i.e E_2 =E but preserving their routes
11. Return graph $D(V_2,E_2)$

Algorithm 4 - Layout Graph Representation

In order to make the ANHOF Method as general as possible the layout of the models should be described in a language. Algorithms are given below that layout each common model graph (step 5d in **LayoutRepresentation** (Algorithm 4)). The

algorithms are based on a general tree automatic graph layout algorithm that positions the father vertices central over its sons and makes the flow of the graph go across the plane. Also the algorithms assume that the origin (coordinates (0,0)) is in the top left of the plane.

4.3.3.1 Basic Algorithms

The general tree automatic graph layout algorithm that lays out a graph from the origin across the page is described in this section, together with variations of the algorithm that layout a graph from the set position of the father vertex. In addition, a variation for use in a Fan In model, where vertices are laid out centred above the fan in vertices is described.

LayoutSubgraph (Algorithm 5) shows how a hierarchical call graph G is laid out so that the flow is across the page. The algorithm is based on a depth first traversal of the hierarchy. The algorithm starts from a given father vertex and recursively traverses the graph. Every time it goes down a level in the tree a set spacing is added to the current x coordinate (*CurrentXCoord*), and every time it goes up a level the set spacing is removed. When a leaf vertex (a fan out value of zero) is reached it stops, marks the vertex as visited and gives it the coordinate ((*CurrentXCoord, CurrentYCoord)*). A set spacing is added to *CurrentYCoord* and the algorithm goes back up a level and traverses the next child. When all the children of the father are traversed the father is marked as visited and centred over its children. This process is repeated until every vertex is visited.

In order to traverse it, the father vertex of all the vertices is needed (*CurrentVertex*) together with the origin ((*CurrentXCoord, CurrentYCoord)*) of the Box that will enclose the graph such that all vertices will be placed to the right of this origin. The vertices will be laid out so that they are *SpacingX* apart on the x plane and *SpacingY* apart on the y plane. When the algorithm is finished the coordinate of last father vertex laid out is ($X_{(\text{Returned})}$, $Y_{(\text{Returned})}$). The coordinate of the bottom right hand corner of the bounding Box is given as ($MaxX_{(\text{Returned})}$, $MaxY_{(\text{Returned})}$). The height of the last vertex is returned in $LastHeight_{(\text{Returned})}$.

LayoutSubgraph (*CurrentVertex, CurrentXCoord, CurrentYCoord, SpacingX, SpacingY , G*)

Where *CurrentVertex* ∈ V, *CurrentXCoord* ∈ ℕ, *CurrentXCoord* ∈ ℕ, *SpacingX* ∈ ℕ, *SpacingY* ∈ ℕ, and *G* = (V,E)

1. Set *MaxX*=0
2. Set *MaxY*=0
3. Set *CurrentVertex* Visited
4. If *CurrentVertex* is a child vertex (fan out value = 0)
    a. Set X Coordinate of CurrentVertex = *CurrentXCoord*
    b. Set Y Coordinate of *Current Vertex* = *CurrentYCoord*
    c. If *CurrentXCoord* + Width of *CurrentVertex* > *MaxX* then set *MaxX*= *CurrentXCoord* + Width of *CurrentVertex*
    d. If *CurrentYCoord* + Height of *CurrentVertex* > *MaxY* then set *MaxY*= *CurrentYCoord* + Height of *CurrentVertex*
5. else
    a. Set *FirstVertex* = 1
    b. Set *NextVertex* = Search *G* for first unvisited vertex from *CurrentVertex*
    c. **LayoutSubgraph**(*NextVertex*, *CurrentXCoord* + *SpacingX* + Width of *CurrentVertex*, *CurrentYCoord*, *SpacingX*, *SpacingY*, *G*)
    d. If $MaxY_{(Returned)}$ > *CurrentYCoord* then set *CurrentYCoord* = $MaxY_{(Returned)}$ + *SpacingY*
    e. else *CurrentYCoord*= $Y_{(Returned)}$ + *SpacingY*
    f. If $MaxX_{(Returned)}$ > *MaxX* then set *MaxX*=$MaxX_{(Returned)}$
    g. *MaxY*=$MaxY_{(Returned)}$
    h. *LastHeight* = $LastHeight_{(Returned)}$
    i. If *FirstVertex*=1
        i. *FirstVertex*=0
        ii. *TopofCurrentTree* = $Y_{(Returned)}$
    j. Set *NextVertex* to next unvisited vertices from *CurrentVertex*

k. Repeat steps c to j until no more unvisited vertices

l. If *FirstVertex*=0

   i. Set Y Coordinate of *CurrentVertex* = (*CurrentYCoord* – *TopofCurrentTree-LastHeight*)/2

   ii. Set X Coordinate of *CurrentVertex* = *CurrentXCoord*

m. else

   i. Set X Coordinate of *CurrentVertex* = *CurrentXCoord*

   ii. Set Y Coordinate of *CurrentVertex* = *CurrentYCoord*

   iii. If *CurrentXCoord* + Width of *CurrentVertex* > *MaxX* then set *MaxX*= *CurrentXCoord* + Width of *CurrentVertex*

   iv. If *CurrentYCoord* + Height of *CurrentVertex* > *MaxY* then set *MaxY*= *CurrentYCoord* + Height of *CurrentVertex*

6. Set $X_{(Returned)}$ = X Coordinate of *CurrentVertex*
7. Set $Y_{(Returned)}$ = Y Coordinate of *CurrentVertex*
8. Set $LastHeight_{(Returned)}$ = Height of *CurrentVertex*
9. Set $MaxX_{(Returned)} = MaxX$
10. Set $MaxY_{(Returned)}=MaxY$
11. Return $(X_{(Returned)}, Y_{(Returned)})$
12. Return $(MaxX_{(Returned)}, MaxY_{(Returned)})$
13. Return $LastHeight_{(Returned)}$
14. Return *G*

Algorithm 5 - The main automatic graph layout algorithm

LayoutSubGraphFromMiddle (Algorithm 6) shows how a graph *G* is laid out so that the father vertex occupies a given coordinate. It has four steps. First of all the height of the graph is calculated, this is done by laying graph *G* out once using **LayoutSubGraph** (Algorithm 5) where the height is returned as $MaxY_{(Return)}$ if the starting origin is (0,0). Next the mid y coordinate is calculated using $MaxY_{(Return)}$ / 2. Next the vertices are marked unvisited so that graph *G* can be laid out properly using **LayoutSubGraph** (Algorithm 5), the starting origin is (given X coordinate, given Y coordinate – mid y coordinate). Graph *G* is returned with the new positions of the vertices.

LayoutSubGraphFromMiddle(*CurrentVertex, StartXCoord,StartYCoord, SpacingX, SpacingY, G*)

Where *CurrentVertex* ∈ V, *StartXCoord* ∈ ℕ, Start*XCoord* ∈ ℕ, *SpacingX* ∈ ℕ, *SpacingY* ∈ ℕ and *G* = (V,E)

1. **LayoutSubgraph**(*CurrentVertex*,0,0,*SpacingX, SpacingY,G)*
2. *MidHeight*=($MaxY_{(Returned)}$ / 2)
3. Unvisit all vertices
4. **LayoutSubgraph**(*CurrentVertex,StartXCoord,StartYCoord-MidHeight,SpacingX, SpacingY,G)*
5. Return *G*

Algorithm 6 - How a tree is laid out with a known mid point

If a vertex has a fan in value and no fan out value then **LayoutSubgraph** (Algorithm 5) will not layout the vertex. This is because of steps 5b and 5j where the next vertex that goes from the current one is actively sought. **LayoutFanInSubGraph** (Algorithm 7) is similar to **LayoutSubgraph** (Algorithm 5) except that it is designed so that the next vertex that is sought in these steps is the first unvisited vertex that has an edge that goes to the current vertex. It is still based on a depth first traversal of the hierarchy and the method remains the same.

LayoutFanInSubGraph (*CurrentVertex, CurrentXCoord, CurrentYCoord, SpacingX, SpacingY, G*)

Where *CurrentVertex* ∈ V, *CurrentXCoord* ∈ ℕ, *CurrentXCoord* ∈ ℕ, *SpacingX* ∈ ℕ, *SpacingY* ∈ ℕ, and *G* = (V,E)

1. Set *MaxX*=0

2. Set *MaxY*=0
3. Set *CurrentVertex* Visited
4. if *CurrentVertex* is a leaf vertex (fan out value = 0)
    a. Set X Coordinate of *CurrentVertex* = *CurrentXCoord*
    b. Set Y Coordinate of *CurrentVertex* = *CurrentYCoord*
    c. If *CurrentXCoord* + Width of *CurrentVertex* > *MaxX* then set *MaxX*= *CurrentXCoord* + Width of *CurrentVertex*
    d. If *CurrentYCoord* + Height of *CurrentVertex* > *MaxY* then set *MaxY*= *CurrentYCoord* + Height of *CurrentVertex*
5. else
    a. Set *FirstVertex* = 1
    b. Set *NextVertex* = Search *G* for first unvisited vertex to *CurrentVertex*
    c. **LayoutFanInSubGraph**(*NextVertex*, *CurrentXCoord* + *SpacingX* + Width of *CurrentVertex*, *CurrentYCoord*, *SpacingX*, *SpacingY*, *G*)
    d. If $MaxY_{(Returned)}$ > *CurrentYCoord* then set *CurrentYCoord* = $MaxY_{(Returned)}$ + *SpacingY*
    e. Else *CurrentYCoord*= $Y_{(Returned)}$ + *SpacingY*
    f. If $MaxX_{(Returned)}$ > *MaxX* then set *MaxX*=$MaxX_{(Returned)}$
    g. *MaxY*=$MaxY_{(Returned)}$
    h. *LastHeight* = $LastHeight_{(Returned)}$
    i. If *FirstVertex*=1
        i. *FirstVertex*=0
        ii. *TopofCurrentTree* = $Y_{(Returned)}$
    j. Set *NextVertex* to next unvisited vertices to *CurrentVertex*
    k. Repeat steps c to j until no more unvisited vertices
    l. If *FirstVertex*=0
        i. Set Y Coordinate of *CurrentVertex* = (*CurrentYCoord* – *TopofCurrentTree-LastHeight*)/2
        ii. Set X Coordinate of *CurrentVertex* = *CurrentXCoord*
    m. else
        i. Set X Coordinate of *CurrentVertex* = *CurrentXCoord*
        ii. Set Y Coordinate of *CurrentVertex* = *CurrentYCoord*
        iii. If *CurrentXCoord* + Width of *CurrentVertex* > *MaxX* then set *MaxX*= *CurrentXCoord* + Width of *CurrentVertex*

iv. If *CurrentYCoord* + Height of *CurrentVertex* > *MaxY* then set *MaxY*= *CurrentYCoord* + Height of *CurrentVertex*

6. Set $X_{(Returned)}$ = X Coordinate of *CurrentVertex*
7. Set $Y_{(Returned)}$ = Y Coordinate of *CurrentVertex*
8. Set $LastHeight_{(Returned)}$ = Height of *CurrentVertex*
9. Set $MaxX_{(Returned)} = MaxX$
10. Set $MaxY_{(Returned)}=MaxY$
11. Return $(X_{(Returned)}, Y_{(Returned)})$
12. Return $(MaxX_{(Returned)}, MaxY_{(Returned)})$
13. Return $LastHeight_{(Returned)}$
14. Return *G*

Algorithm 7 - How vertices that have a fan in value are laid out

4.3.3.2 Common Model Graph Automatic Graph Layout Algorithms

Given below are the automatic graph layout algorithms for the common model graphs described above. All the fixed common model graphs use **LayoutSubgraph** (Algorithm 5), **LayoutSubGraphFromMiddle** (Algorithm 6) or **LayoutFanInSubGraph** (Algorithm 7) to layout the vertices involved in each common model graph. This is because they are defined in terms of the fan in or fan out properties of the vertices. All the algorithms below require spacing between vertices (*SpacingX* and *SpacingY).* The graph to which the algorithm applies to is given in the parameter *G.* In all the automatic graph layout algorithms below the vertices and edges are sorted. There are many methods of sorting them. For example, one is to sort the vertices in alphabetical order and the edges into another order so that the vertices with which they are associated are also in alphabetical order. Another is to sort the vertices into fan out order, so that the one with the highest fan out properties are at the top of the list. A third is the Topological Sort. These and others are be evaluated in future chapters. The sorting is carried out because **LayoutSubgraph** (Algorithm 5) finds the first unvisited vertex in steps 5b and k and the first unvisited vertex is the first in a queue of vertices and edges, hence the need to sort the queue.

LayoutFanOutModel (Algorithm 8) shows how a Fan Out common model graph is laid out. In the common model graph there is one father vertex, a vertex with the fan in property of zero. Common model graph G is passed through the **LayoutSubgraph** (Algorithm 5) with this vertex as the start vertex. Common model graph M is then returned with the correct layout of the vertices.

LayoutFanOutModel (*SpacingX*, *SpacingY*, *M*)

Where $SpacingX \in \mathbb{N}$, $SpacingY \in \mathbb{N}$, and $M = (V,E)$

1. Sort V
2. Sort E
3. Set *FatherVertex* = Vertex with fan in value of 0
4. **LayoutSubGraph**(*FatherVertex*,0,0,*SpacingX*,*SpacingY*,*M*)
5. Return *M*

Algorithm 8- The automatic graph layout algorithm for a Fan Out common model graph

LayoutFanInModel (Algorithm 9) shows how a Fan In common model graph is laid out. In the common model graph there is one father vertex, a vertex with the fan out property of zero. Common model graph M passes through the **LayoutFanInSubGraph** (Algorithm 7) with this vertex as the start vertex. Common model graph M is then returned with the correct layout of the vertices.

LayoutFanInModel (*SpacingX*, *SpacingY*, *M*)

Where $SpacingX \in \mathbb{N}$, $SpacingY \in \mathbb{N}$, and $M = (V,E)$

1. Sort V
2. Sort E
3. Set *FatherVertex* = Vertex with fan out value of 0
4. **LayoutFanInSubGraph**(*FatherVertex*,0,0,*SpacingX*,*SpacingY*,*M*)
5. Return *M*

Algorithm 9 - The automatic graph layout algorithm for a Fan In common model graph

LayoutSplit1Model (Algorithm 10) describes how a Split 1 common model graph is laid out. The father vertices are found, the vertices that have a fan in value of zero. If this is a valid Split1 common model graph then there should be exactly two of these vertices. Take the first of these vertices and pass it through **LayoutSubgraph** (Algorithm 5). Then the second of the vertices should be given the coordinate ($X_{(returned)}$ + X spacing, $Y_{(returned)}$).

LayoutSplit1Model (*SpacingX*, *SpacingY*, *M*)

Where *SpacingX* ∈ ℕ, *SpacingY* ∈ ℕ, and *M* = (V,E)

1. Sort V
2. Sort E
3. Set *TopLevel* = Top Level Vertices (vertices with fan in = 0) in Graph *M*
4. | *TopLevel* | = 2
5. *CurrentVertex* = Take first member *TopLevel*
6. **LayoutSubGraph**(*CurrentVertex*,0,0,*SpacingX*, *SpacingY*, *M*)
7. *CurrentVertex* = The next member of *TopLevel*
8. Set X Coordinate of *CurrentVertex* = $X_{(returned)}$ + SpacingX
9. Set Y Coordinate of *CurrentVertex* = $Y_{(returned)}$
10. Return *M*

Algorithm 10 - The automatic graph layout algorithm for a Split 1 common model graph

LayoutSplit2Model (Algorithm 11) shows how a Split 2 common model graph is laid out. The father vertices are found, the vertices that have a fan in value of zero. If this is a valid Split 2 common model graph then there should be exactly two of these vertices. Take the father vertex that has the lowest fan out value and lay it out using **LayoutSubgraph** (Algorithm 5). Use the other father vertex and lay it out using **LayoutSubGraphFromMiddle** (Algorithm 6) with the starting point being ($X_{(returned)}$,$Y_{(returned)}$).

LayoutSplit2Model (*SpacingX*, *SpacingY*, *M*)

Where *SpacingX* ∈ ℕ, *SpacingY* ∈ ℕ, and *M* = (V,E)

1. Sort V
2. Sort E
3. Set *TopLevel* = Top Level vertices (vertices with fan in = 0) in Graph *M*
4. Set *MiddleVertex* = Top Level vertex with largest fan out value
5. Set *LeftVertex* = Other member of *TopLevel*
6. **LayoutSubGraph**(*LeftVertex*,0,0,*SpacingX*,*SpacingY*, *M*)
7. **LayoutSubGraphFromMiddle**(*MiddleVertex*, $X_{(returned)}$,$Y_{(returned)}$, *SpacingX*,*SpacingY*, *M*)
8. Return *M*

Algorithm 11 - The automatic graph layout algorithm for a Split 2 common model graph

LayoutSplit3Model (Algorithm 12) shows how a Split 3 common model graph is laid out. The middle vertex is found, a vertex with both a fan out and fan in value, there should only be one of these. Pass this through **LayoutFanInSubGraph** (Algorithm 7) and then **LayoutSubGraphFromMiddle** (Algorithm 6) with the centre point being ($X_{(returned)}$, $Y_{(returned)}$). Common model graph *M* is then returned with the correct layout.

LayoutSplit3Model (*SpacingX*, *SpacingY*, *M*)

Where *SpacingX* ∈ ℕ, *SpacingY* ∈ ℕ, and *M* = (V,E)

1. Sort V
2. Sort E
3. Set *MiddleVertex* = Vertex with fan out >0 and fan in >0
4. **LayoutFanInSubGraph**(*MiddleVertex*,0,0,*SpacingX*,*SpacingY*)
5. **LayoutSubGraphFromMiddle**(*MiddleVertex*, $X_{(returned)}$,$Y_{(returned)}$, *SpacingX*,*SpacingY*, *M*)
6. Return *M*

Algorithm 12- The automatic graph layout algorithm for a Split 3 common model graph

LayoutChainModel (Algorithm 13) shows how a Chain common model graph is laid out. In the common model graph there is one father vertex, a vertex with the fan in property of zero. Common model graph *M* is passed through the **LayoutSubgraph** (Algorithm 5) with this vertex as the start vertex. Common model graph *M* is then returned with the correct layout of the vertices.

LayoutChainModel (*SpacingX*, *SpacingY*, *M*)

Where $SpacingX \in \mathbb{N}$, $SpacingY \in \mathbb{N}$, and $M = (V,E)$

1. Sort V
2. Sort E
3. Set *StartVertex* = Vertex with fan in value of 0
4. **LayoutSubGraph**(*StartVertex*,0,0,*SpacingX*,*SpacingY*,*M*)
5. Return *M*

Algorithm 13 - The automatic graph layout algorithm for a Chain common model graph

LayoutChainToFanOutModel (Algorithm 14) shows how a Chain To Fan Out common model graph is laid out. In the common model graph there is one father vertex, a vertex with the fan in property of zero. Common model graph *M* is passed through the **LayoutSubgraph** (Algorithm 5) with this vertex as the start vertex. Common model graph *M* is then returned with the correct layout of the vertices.

LayoutChainToFanOutModel (*SpacingX*, *SpacingY*, *M*)

Where $SpacingX \in \mathbb{N}$, $SpacingY \in \mathbb{N}$, and $M = (V,E)$

1. Sort V
2. Sort E
3. Set *StartVertex* = Vertex with fan in value of 0
4. **LayoutSubGraph**(*StartVertex*,0,0,*SpacingX*,*SpacingY*,*M*)
5. Return *M*

Algorithm 14 – The automatic graph layout algorithm for a Chain to Fan Out common model graph

LayoutTriangleModel (Algorithm 15) shows how a Triangle common model graph is laid out. There should be three vertices in the common model graph. The father vertex, a vertex with a fan in equal to zero, is given the position nearest the left hand edge of the plane, (0,0). The vertex with a fan in and fan out value of one is positioned a set spacing to the right of the father vertex, and the other vertex is positioned a set spacing underneath this vertex.

LayoutTriangleModel (*SpacingX*, *SpacingY*, *M*)

Where $SpacingX \in \mathbb{N}$, $SpacingY \in \mathbb{N}$, and $M = (V,E)$

1. Set *TopLevel* = Top Level vertices (vertices with fan in of 0) in Graph *M*
2. Set X Coordinate of *TopLevel* =*0*
3. Set Y Coordinate of *TopLevel* =*0*
4. Set *CurrentVertex* = Vertex in *TopLevel* with fan in = 1 and fan out = 1
5. Set X Coordinate of *CurrentVertex* =*SpacingX* + Width of *TopLevel*
6. Set Y Coordinate of *CurrentVertex* =*0*
7. Set *CurrentVertex* = Last Vertex from *TopLevel*
8. Set X Coordinate of *CurrentVertex* =*0*
9. Set Y Coordinate of *CurrentVertex* =*SpacingY* + Height of *TopLevel*
10. Return *M*

Algorithm 15 - The automatic graph layout algorithm for a Triangle common model graph

BoxModelLayout(Algorithm 16) shows how a Box common model graph is laid out. It is difficult to detect where vertices are located because three of the vertices have the same fan in and fan out properties. If the positions are wrong then edges cross in the centre of the layout, whereas a correct layout will be planar. Three vertices have the same fan in and fan out properties, but the bottom right vertex has an individual fan in and fan out property. The top left vertex is the vertex that flows to the top right vertex and then to the bottom right vertex that has an individual fan in and fan out property. The bottom left vertex is the one that is left and is the one that flows to the top left and bottom right vertices. All the vertices should be aligned with the left most point on the

vertex above it and horizontally aligned with the top most point on the vertex next to it. Therefore the vertex in the top left corner of the plane is placed at the origin (0,0), the bottom left vertex is placed a set spacing below, the top right is placed a set spacing to the right and the bottom right vertex is placed a set spacing below.

BoxModelLayout(*SpacingX*, *SpacingY*, *M*)

Where $SpacingX \in \mathbb{N}$, $SpacingY \in \mathbb{N}$, and $M = (V,E)$

1. Set *BottomRight* = Vertex with fan in of 2 in *M*
2. Mark *BottomRight* as visited
3. Set *TopRight* = The vertex that flows to *BottomRight*
4. Set *TopLeft* = The vertex that flows to *TopRight* and then to *BottomRight*
5. Mark *TopRight* as visited.
6. Mark *TopLeft* as visited.
7. Set *BottomLeft* = unvisited vertex that flows to *TopLeft* and *BottomRight*
8. Set *MaxWidth* = Greatest Width of *TopLeft* and *BottomLeft*
9. Set *MaxHeight* = Greatest Height of *TopLeft and TopRight*
10. Set X Coordinate of *TopLeft* = 0
11. Set Y Coordinate of *TopLeft* = 0
12. Set X Coordinate of *BottomLeft* = 0
13. Set Y Coordinate of *BottomLeft* = *MaxHeight* + *SpacingY*
14. Set X Coordinate of *TopRight* = *MaxWidth* + SpacingX
15. Set Y Coordinate of *TopRight* = 0
16. Set X Coordinate of *BottomRight* = *MaxWidth* + *SpacingX*
17. Set Y Coordinate of *BottomRight* = *MaxHeight* + *SpacingY*
18. Return *M*

Algorithm 16 - The automatic graph layout algorithm for a Box common model graph

4.3.3.3 Layout Whole Graph

In order to perform step 9 of **LayoutRepresentation** (Algorithm 4) an automatic graph layout algorithm is required. The ANHOF method is designed to layout call graphs that

are hierarchical / tree like structures. There are many algorithms that could be used to perform a layout of such a structure. Which algorithm produces the best graph in terms of its aesthetics is subject to evaluation. **GraphLayout** (Algorithm 17) suggests an algorithm that uses **LayoutSubgraph** (Algorithm 5) to layout the structure. Its performance will be evaluated against standard algorithms in future chapters. Again the vertices and edges are sorted using methods discussed earlier. The father vertices are found, the vertices that have a fan in value of 0, these are then laid out one at a time using **LayoutSubgraph** (Algorithm 5).

This algorithm is a variation of the Graph Tool Automatic graph layout algorithm given in Bodhuin [16] , it is used in later versions of Graph Tool that are implemented by Young [180], and is the one used in future uses of the Graph Tool algorithm in conjunction with the ANHOF method.

GraphLayout (*SpacingX*, *SpacingY*, *G*)

Where $SpacingX \in \mathbb{N}$, $SpacingY \in \mathbb{N}$, and $G = (V,E)$

1. Sort V
2. Sort E
3. Set *TopLevel* = *TopLevel* vertices (vertices with fan in of zero) in Graph *G*
4. Set *CurrentYPosition* = 0
5. Sort *TopLevel*
6. For each member of *TopLevel:* -
    a. Set *CurrentTLVertex* = Fist member of *TopLevel*
    b. **LayoutSubGraph**(*CurrentTLVertex*, 0, *CurrentYPostion*, *SpacingX*, *SpacingY*, *G*)
    c. Set *CurrentYPosition* = $MaxY_{(Returned)}$ + *SpacingY*
    d. Set *CurrentTLVertex* = next unvisited member of *TopLevel*
7. Return *G*

Algorithm 17 -Shows an algorithm that will layout a hierarchical graph

4.3.4 Graph Display System

The graph can be displayed on a computer display, on paper or both. Chapter 3 gives many display tools that can be used to display the graph. This part of the ANHOF method is performed using one of them.

4.4 Summary

In this chapter the ANHOF method of graph layout for call graphs is given. The common structures that are present in call graphs have been identified and their respective layouts described. Each common model graph is discussed in turn giving their aesthetics, layout and definition.

The ANHOF method is a four-part process, consisting of three processing parts and one output part. In the first part, the Graph Isomorphism System, the common model graphs are detected. The Match Analyser then sorts the matches to the common model graphs. The valid matches are then laid out along with the rest of the graph using their associated automatic graph layout algorithms and standard algorithms in the Graph Layout System. When all this is performed the graph is displayed using one of the many graph display systems available. In the next chapter an implementation of the ANHOF method is given, this is known as the ANHOF system.

5. Implementing the ANHOF Method

5.1 Introduction

The ANHOF system is a four part process that implements the ANHOF method. The architecture of the ANHOF system follows that of the ANHOF method given in Figure 43. This chapter describes the implementation of the ANHOF.

5.2 The ANHOF System

In Chapter 4, a four part process is presented for laying out call graphs, called the ANHOF method. It consists of three processing steps and one display step. The Graph Isomorphism System detects all the valid and invalid matches to the given common model graphs. These are then sent to the Match Analyser whose job is to remove all the invalid matches, producing a list of valid matches and a representation of a graph in terms of the common model graphs present. Then the specification of the required layout, in terms of its aesthetics, is used to layout the representation using standard, customised or common model graph automatic graph layout algorithms. This produces the input file for a graph display system.

The ANHOF system is implemented on a PC PII 450 MHz running Windows 95 with 128 MB of memory. This provides ample space to store and layout the size of graphs (greater than 150 vertices) required and provides the graphical ability to display them. The C++ program used was Microsoft Visual C++ version 6.0. SWI Prolog version 3.2.8 [173] was used to provide logic processing of lists.

5.2.1 Graph Isomorphism System

This is the implementation of the first part of the ANHOF method known as the Graph Isomorphism System. It finds both the fixed and variable common model graphs in the call graph, and all the valid and invalid matches to the common model graphs. **SearchForModelGraph** (Algorithm 2) shows detection of these is a three step process,

one step to detect the fixed common model graphs and one to gather all the information needed to detect the variable common model graphs in the final step. This equates to two major and one simple search step. The two major processing steps equate to the searching for the two types of common model graphs present in a call graph, variable and fixed. When implementing these in the ANHOF system two programs are required, known as the Fixed Model Detection System and the Variable Model Detection System. The outline of the Graph Isomorphism System is given in Figure 44.

Graph input into the ANHOF system is in the form of the Graph Tool GIN input format. This basically lists all the vertices and edges and therefore provides a simple input format to use. Details of the file format are given in Appendix 2. To use this file format processing is necessary to obtain the fan information that is used to detect the variable common model graphs.

Fixed common model graph detection is performed using one of the algorithms presented in Chapter 3. There are many implementations of the different algorithms of graph and subgraph isomorphism. One such implementation is provided by Messmer [114] and is known as the Graph Matching Toolkit. Here, Messmer implements Ullman's Algorithm, his own decision tree approach and other tree searching algorithms. The toolkit is written in C++ and so integrates well with the other programs. This implements steps 1a to c of **SearchForModelGraph** (Algorithm 2) and is know as the Fixed Model Detection System.

Variable common model graphs are detected by list processing using logic programming. The ANHOF system uses Prolog [34]. Prolog's resolution based search strategy provides an efficient method of searching the various vertices, edges and fan information fact bases to implement step 3a to d of **SearchForModelGraph** (Algorithm 2). Each variable common model graph is implemented as a single Prolog rule and executed singly on each fact base. Both common model graph detection programs output a fact base of matches. A description of the graph fact base, the fan information fact base and the match output fact base can be found in Appendix 3. The group of Prolog rules is collectively known as the Variable Model Detection System.

In Chapter 4 it is suggested that the common model graphs in the graph be described in some form of language. This will allow common model graphs to be given in an easy to understand form and allow new common model graphs to be described or an existing one to be amended. In the ANHOF system this is achieved in two different ways, one for the fixed common model graphs and one for the variable common model graphs. Fixed common model graphs have a fixed number of vertices and edges and can therefore be described as an adjacency matrix. The adjacency matrix definition of the fixed common model graphs present in call graphs is given in the Appendix 4. Variable common model graphs can consist of any number of vertices but have a common edge structure. The definitions of the variable common model graphs are best given as subsets of the sets of edges and vertices. In the ANHOF system these are processed using logic rules in Prolog. The rules provide a simple method of describing a common model graph in terms of its vertices and edges and other information about the graph for instance the fan in and fan out information. To aid in the description of a common model graph standard routines are provided for output of the various fact bases, details are provided in the Appendix 3.

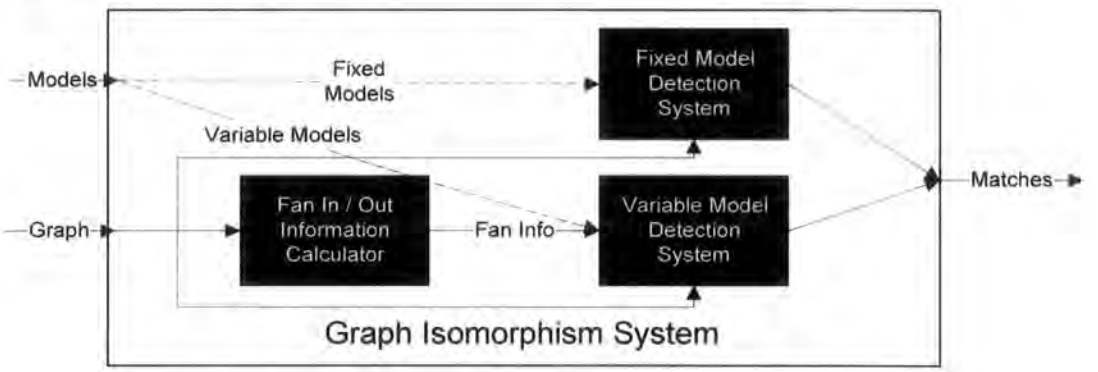


Figure 44 - The Graph Isomorphism System

5.2.2 Match Analyser

The Match Analyser implements **AnalysisOfMatches** (Algorithm 3). It filters the matches to the common model graphs found by the Graph Isomorphism System, removing the invalid ones. In the ANHOF method an invalid match is one that involves a vertex that is part of another common model graph. This kind of filtering is again easily performed using Prolog. The Match Analyser processes the match fact base from

the Graph Isomorphism System and the graph fact base, creating another match fact base populated with the valid matches. It also creates the representation discussed in Chapter 4. Details of this representation and the match and graph fact base can be found in Appendix 3. It operates on a first come first served basis, where if the first match involved all the vertices in the graph then all the others would be invalid.

5.2.3 Graph Layout System

Here the automatic graph layout algorithm is performed in accordance with the aesthetics, such as the vertex spacing, detailed by the user in an input file. Standard automatic graph layout algorithms are used to layout the graphs, therefore it is necessary to provide a large set of algorithms so that any of the standard algorithms discussed in Chapter 2 can be used. There are several algorithm libraries available ([28], [49] etc). Most of them however implement algorithms for certain classes of graph. However a combination of Library of Efficient Data Algorithms (LEDA) [111] and Algorithms for Graph Drawing (AGD) [119] provides implementations of all the algorithms discussed in Chapter 2. These libraries are imported into C++ very easily, they are easily extended to implement the algorithms given in Chapter 4 for laying out the common model graphs and to implement **LayoutSubgraph** (Algorithm 5), **LayoutSubGraphFromMiddle** (Algorithm 6) and **LayoutFanInSubGraph** (Algorithm 7). AGD also provides a language for describing aesthetics, the language is used as an input into the AGD Server, which is a program that allows a graph to be imported and laid out using one of the AGD library of algorithms. Details of this language can be found in [87]. The general structure of Graph Layout System for the ANHOF system is given in Figure 45. It is written in C++ and implements **LayoutRepresentation** (Algorithm 4).

In Chapter 4 it is suggested that the automatic graph layout algorithms for the common model graphs are described in a simple language. This will allow future expansion to the library of common model graphs and for the method to be used on other types of graph. In the ANHOF system this is done using C/C++ with AGD/LEDA extensions. AGD/LEDA allows its own library of automatic graph layout algorithms to be expanded with user implemented ones. In order to do this it has made certain classes available for

allocating positions to vertices, routing edges etc. Also it imposes a certain style on the program file that the automatic graph layout algorithm is programmed in. Experience is required to implement a new automatic graph layout algorithm. However it does allow new automatic graph layout algorithms to be programmed in a readable fashion and these are easily incorporated into the Graph Layout System, this however will need recompiling every time a new automatic graph layout algorithm is implemented. Details of how to program a automatic graph layout algorithm can be found in [111] and [87].

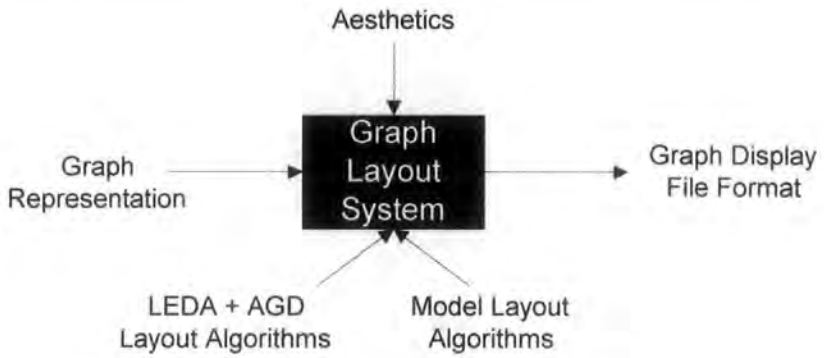


Figure 45 - The Graph Layout System

5.2.4 Graph Display System

Chapter 3 shows there are many graph display tools available. It is required that the graph display tool in the ANHOF System has primarily a simple file input that is easy to read. Secondly the input format allows vertices to be given Cartesian coordinates and if possible allow routes to be given to edges. These requirements reduced the choice drastically. Two main ones used, VCG [146] and University of Durham's own Graph Tool. Unfortunately neither really allow routes to be given to edges, Graph Tool does allow one kink in the line but it is unreliable. Both the file formats are equally easy to read and allow edges to be positioned. The original intention of the project was to improve the Graph Tool layout; therefore the ANHOF system remains loyal to its routes and uses Graph Tool to display the final graph layout.

5.3 Summary

Above is a short description of how a prototype of the ANHOF method of call graph layout is implemented in a system known as the ANHOF system. It has shown that it can be implemented in a four-part program, made up of three processing parts and one display part. Each step has been described giving details of the languages in which they were implemented and any outside programs that were used. Details of languages that are used to describe the graph aesthetics, the automatic graph layout algorithms for the common model graphs and the structure of the common model graph are also given.

It has shown that variable common model graphs are described as Prolog rules, which are then applied to fact bases about the graph, allowing variable common model graphs to be detected in the Graph Isomorphism System. Fixed common model graphs are described as an adjacency matrix and detected by one of the algorithms in the Messmer's Graph Matching Toolkit. The chapter has shown that the matches to the common model graphs that are found are filtered by a Prolog implementation of the Match Analyser. The layout of the graph is obtained using the Graph Layout System, which is a C++ program using LEDA and AGD to provide standard automatic graph layout algorithms. The common model graph automatic graph layout algorithms are also programmed using the C++ / LEDA / AGD combination and the final layouts are displayed on the Graph Tool graph display system.

The next three chapters will describe the optimisation of the ANHOF system. The layouts it produces will be compared with other layouts from tools and algorithms and the metrics of the layout will be assessed.

6. Tuning the ANHOF System

In Chapter 4 a method of laying out call graphs is presented, called the ANHOF method. In Chapter 5 an implementation of it is discussed, called the ANHOF system. In order to use the ANHOF system to gain 'better' call graphs it is necessary to customise the various settings of its components. In the following chapter the lengths of chains, and the levels of fan in and fan out in the Variable Model Detection System are discussed, together with the selection of the best isomorphism algorithm for the fixed models. The maximisation of the number of models produced by the Match Analyser is discussed. Finally the sort orders of the vertices to reduce the edge crossings in the final layout is presented.

6.1 Graph Isomorphism System

The Graph Isomorphism System is the part of the ANHOF system that detects all the matches of the models in the graph. There are two types of models in the graph, Fixed and Variable, each having a separated system to detect them. The settings necessary to detect the maximum number of models are detailed below.

6.1.1 Variable Model Detection System

Chain, fan in and fan out settings are used to specify the minimum number of vertices that need to be present in order to detect the various variable models. They are the parameters that are discussed in Chapter 4. Too large a setting means that few models are found and the performance of the Variable Model Detection System suffers by slowing it down. Too small a setting means that a large number of common model graphs are detected, effecting performance of the Match Analyser and Variable Model Detection System and preventing the proper filtering of models by the Match Analyser system.

Five graphs with between approximately 10 and 100 vertices and 10 and 300 edges are selected from the approximately 250 call graphs of the GCC version 2.58 compiler. The properties and names of the graphs are given in Table 12. First of all the common model graphs are detected manually by searching a layout of each obtained from Graph Tool's automatic graph layout algorithm. The graphs are then tested with the Graph Isomorphism System with relatively high settings for the Variable Model Detection System to check the output is correct. When the results of detecting the common model graphs manually and by computer are compared it is found that whilst they largely matched, if the settings were lower the results would match better. Also when the computer output was run through the Match Analyser it filtered less valid matches and took a similar time than later trials. The eventual layout had few common structures present. After lowering the settings a few times it was found that it was best to obtain the maximum matches possible because the Match Analyser executes in an acceptable time. The resulting layout has the most common structures present and the aesthetics and metrics were better.

Graph Name	Vertices	Edges
cp-search	9	8
genopinit	21	22
varasm	26	32
recog	52	70
real - 2	111	319

Table 12 - The properties of the chosen graphs

It is ascertained that the settings in Table 13 produced maximum performance of the Variable Model Detection System and provide an improved graph at the end in terms of the resulting graphs aesthetics and metric quality. The settings for the fan out and fan in were sufficiently low enough to detect most if not all of the relevant matches, but prevent any models that are better dealt with by a standard automatic graph layout algorithm.

Setting	Level
Fanoutlevel	3
Faninlevel	3
Chainlength	3
Chainfanoutlevel	2
Lengthofchain	2
Commonfanoutnumber	3
Commonsplit2fanoutlevel	3
Split2fanoutlevel	3
Split3faninlevel	2
Split3fanoutlevel	2

Table 13 - The settings for the various model detect systems

6.1.2 Fixed Model Detection System

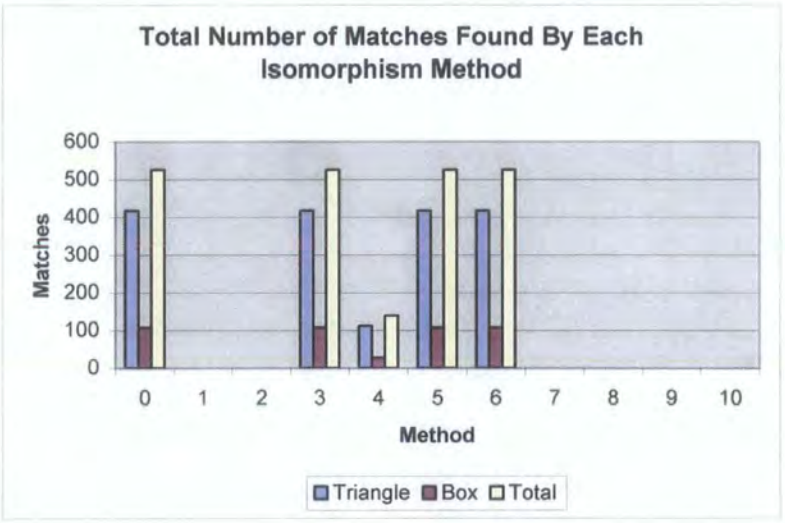
Fixed common model graphs are detected differently to variable models. The fixed structure enables graph isomorphism techniques to be used. In order to perform this a toolkit of many graph isomorphism algorithms is used. In particular it uses Messmer's [114] Graph Matching toolkit. This uses a variety of algorithms in order to detect the models present in the graph. In order to use the toolkit it is necessary to evaluate which isomorphism algorithm is the most successful. A successful algorithm is one that detects the most correct models in the graph.

Three graphs taken from the GCC version 2.58 compiler are passed through the various isomorphism algorithms present in the toolkit. Table 14 shows the properties of the graphs. The total matches to the fixed model graphs obtained by using the various algorithms present are calculated. Figure 46 shows the result of this. It shows that the algorithms given in Chapter 3 (methods 0, 5 and 6) and exact decomposition (method 3) work the best, detecting the same number of matches (526 model matches). Figure 46 shows that some algorithms did not work on the graphs at all.

Tests on other not completely connected graphs using the implementations of the decision tree method of isomorphism (methods 5-10) it finds that that the input graph did not meet many of the algorithm's pre-processing conditions. This is because the graph is connected. Call graphs of code often are not connected together because the code has been parsed in the various sub modules. Therefore a procedure may exist on its own in that module but be called in another module and a base module is likely to consist of many unconnected procedures. Whilst it is possible to circumvent this it is deemed to be undesirable, especially when certain algorithms allow this.

Graph Name	Vertices	Edges
combine2-1	112	300
c-decl2	151	295
tmp	219	675

Table 14- The graph properties used to test the isomorphism algorithms



Method Key	
0	Ullman
1	A*
2	A* with lookahead
3	Exact Decomposition
4	Inexact Decomposition
5	Complete decision tree
6	k-depth pruned complete decision tree
7	Breadth pruned decision tree
8	Both breadth and k-depth pruned decision tree
9	Breadth-pruned decision tree (single model)
10	Both breadth- and k-depth pruned decision tree (single model)

Figure 46 - The performance of various isomorphism algorithms

Many of the methods were unsuitable for the purpose. The ANHOF system has clearly defined common model graphs, the graph structure is known and the graph to be searched for in the graph is clearly defined. Therefore algorithms that search for

common model graphs and the entire subgraph variations of that common model graph are not required. Therefore the algorithms known as A* (method 1), A* with look ahead (method 2), and inexact decomposition matching (method 4) are not needed. Indeed they should not even work. For details about the algorithms see [114].

The other methods are variations on the decision tree approach by pruning the decision tree. They are largely experimental and therefore will not be used. The Graph Matching Toolkit has implemented the pruning techniques as graph isomorphism algorithms and, therefore, compare one graph with another. The ANHOF system requires implementations of subgraph isomorphism algorithms because the common model graphs are smaller than the main graphs and comparing the graphs is not going to work. The ANHOF system is looking for occurrences of the common model graphs in the main graph, these techniques are known as subgraph isomorphism techniques. Therefore methods 7-10 will not produce any matches to the common model graphs. For a more thorough performance evaluation see [114].

When searching for the models in the graphs the labels and any identifiable features of vertices can be disregarded and therefore the graphs become unlabelled. Messmer's [114] thesis suggests that for unlabelled graphs of less than 500 vertices Ullman's algorithm [164] (method 0) is the most suitable. The result in Figure 46 appears to agree with this conclusion. The drawback is that the performance is dependent on the number of common model graphs. To aid in solving this problem a graph will be searched to find one model at a time. In Fixed Model Detection System the implementation of Ullman's algorithm will be used as the algorithm of detecting the common model graphs.

6.2 Match Analyser

The purpose of the Match Analyser is to reduce all the matches to valid ones. Chapter 4 suggested that there are many different definitions of a valid match. In order to avoid problems of vertices being laid out twice because they are in different models, the rule is applied that a model cannot involve a vertex that it is part of another model. The method chosen to do this is on a first come first served basis. Then, if the first match

processed involved all the vertices in the graph then all the other matches would not be valid, however this is rarely the case. It was shown in Chapter 1 that in order to increase comprehension of graphs it is necessary to have many recognizable common structures present in the eventual layout. However when trying to improve the layout in terms of its aesthetics and metrics it may be sometimes best to have many small models present and few large, and thereby maximising the number of common structures present. Conversely it may be best to have few common model graphs present in the layout with most of the vertices involved in one or two models where it is known how to lay them out.

In Chapter 8 the performance of the ANHOF system against standard automatic graph layout algorithms is assessed, however it is only assessed in terms of the metrics of the layout. It is beyond the scope of the research to assess whether comprehension was aided by increasing the number of common model graphs in the eventual layout. In order to do the investigations in Chapter 8 it is necessary to perform analysis to ascertain how to get the maximum number of valid matches out of the Match Analyser. An investigation into how to achieve this is presented below. In order to do further investigations into the performance of the ANHOF system it is necessary to ascertain a method of passing a reasonable number of model matches through the Match Analyser. This order is known as the natural order. The performance of this order is also assessed in Chapter 8, but how to obtain it is investigated below.

6.2.1 Maximising the Number of Valid Matches

In order to increase the comprehension of graphs and generate the best aesthetics it may be necessary to maximise the number of valid model graphs used to layout the graph. To perform this on a 'first come first served' basis it is necessary to find the best order in which to pass the list of matches through the Match Analyser in order to maximised the number of valid matches. An experiment to ascertain this order is described below.

Three graphs were taken from the GCC version 2.58 compiler. They were of varying size and varying number of common model graphs present. The properties of the graphs are given below in Table 15.

Graph Name	Vertices	Edges	Total Matches	Triangles	Box	Split 1	Split 2	Split 3	Chain	FanOut	Fan In	Chain to Fan Out
combine2-1	112	300	716	260	71	194	96	23	0	30	40	2
cdecl-2	151	296	437	37	10	219	101	7	1	29	32	1
real-2	111	391	1120	143	52	544	246	33	1	54	42	5

Table 15 - The properties of the tested graphs

There are many orders in which the models can be sent through the Match Analyser. A list of all the matches to the common model graphs in each of the above graphs is contained in a file in each of the following orders: -

- The matches to the common model graphs placed in the file in random order (random).
- The matches to the common model graphs placed in the file in ascending alphabetical order (ascending whole).
- Take each set of matches to a common model graph and sort it into ascending alphabetical order. Combine all the sets of matches together in a file, a file made of every combination of sets (ascending individual).
- The matches to the common model graphs placed in the file in descending alphabetical order (descending whole)
- Take each set of matches to a common model graph and sort it into descending alphabetical order. Combine all the sets of matches together in a file, a file made of every combination of sets (descending individual).
- Take alternate matches from a file formed by combining the sets of matches together, files are formed from combining the sets of matches together in every combination (every other whole).
- Take alternate matches from each set of matches to a common model graph. Create a similar set for each model graph; combine all the sets together in one file. Form a file in every combination of sets (every other individual).
- Form sets of all matches to the model graphs as the Graph Isomorphism System finds them; combine the sets together in every combination (as they come).

In many of the above file orders a combination of sets is mentioned. This is a combination of the matches to each of the nine common model graphs. Therefore there are 362880 (9!) combinations that the models can be combined in one file. Each combination of sets takes upwards from a minute to create, process and interpret. Therefore the 362880 combinations would take 252 days to process and was obviously impossible; a cross section of the combinations was therefore taken, 72 combinations were chosen. They represented putting the common model graphs that would involve the most vertices (Split 1, Split 2 and Split 3) first. Then the common model graphs that would involve the least vertices (Triangle and Box) first, and, finally other variations. The combinations are given in Table 16.

Combination	Combination Number
Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan In Fan Out	1
Split 1 Triangle Split 2 Split 3 Box Chain to Fan Out Chain Fan In Fan Out	2
Split 1 Split 2 Triangle Split 3 Box Chain to Fan Out Chain Fan In Fan Out	3
Split 1 Split 2 Split 3 Triangle Box Chain to Fan Out Chain Fan In Fan Out	4
Split 1 Split 2 Split 3 Box Triangle Chain to Fan Out Chain Fan In Fan Out	5
Split 1 Split 2 Split 3 Box Chain to Fan Out Triangle Chain Fan In Fan Out	6
Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Triangle Fan In Fan Out	7
Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan In Triangle Fan Out	8
Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan In Fan Out Triangle	9
Triangle Split 2 Split 1 Split 3 Box Chain to Fan Out Chain Fan In Fan Out	10
Triangle Split 2 Split 3 Split 1 Box Chain to Fan Out Chain Fan In Fan Out	11
Triangle Split 2 Split 3 Box Split 1 Chain to Fan Out Chain Fan In Fan Out	12
Triangle Split 2 Split 3 Box Chain to Fan Out Split 1 Chain Fan In Fan Out	13
Triangle Split 2 Split 3 Box Chain to Fan Out Chain Split 1 Fan In Fan Out	14
Triangle Split 2 Split 3 Box Chain to Fan Out Chain Fan In Split 1 Fan Out	15
Triangle Split 2 Split 3 Box Chain to Fan Out Chain Fan In Fan Out Split 1	16
Split 2 Triangle Split 1 Split 3 Box Chain to Fan Out Chain Fan In Fan Out	17
Triangle Split 2 Split 1 Split 3 Box Chain to Fan Out Chain Fan In Fan Out	18
Triangle Split 1 Split 3 Split 2 Box Chain to Fan Out Chain Fan In Fan Out	19
Triangle Split 1 Split 3 Box Split 2 Chain to Fan Out Chain Fan In Fan Out	20
Triangle Split 1 Split 3 Box Chain to Fan Out Split 2 Chain Fan In Fan Out	21
Triangle Split 1 Split 3 Box Chain to Fan Out Chain Split 2 Fan In Fan Out	22
Triangle Split 1 Split 3 Box Chain to Fan Out Chain Fan In Split 2 Fan Out	23
Triangle Split 1 Split 3 Box Chain to Fan Out Chain Fan In Fan Out Split 2	24
Split 3 Triangle Split 1 Split 2 Box Chain to Fan Out Chain Fan In Fan Out	25
Triangle Split 3 Split 1 Split 2 Box Chain to Fan Out Chain Fan In Fan Out	26
Triangle Split 1 Split 3 Split 2 Box Chain to Fan Out Chain Fan In Fan Out	27
Triangle Split 1 Split 2 Box Split 3 Chain to Fan Out Chain Fan In Fan Out	28
Triangle Split 1 Split 2 Box Chain to Fan Out Split 3 Chain Fan In Fan Out	29
Triangle Split 1 Split 2 Box Chain to Fan Out Chain Split 3 Fan In Fan Out	30
Triangle Split 1 Split 2 Box Chain to Fan Out Chain Fan In Split 3 Fan Out	31
Triangle Split 1 Split 2 Box Chain to Fan Out Chain Fan In Fan Out Split 3	32
Box Triangle Split 1 Split 2 Split 3 Chain to Fan Out Chain Fan In Fan Out	33
Triangle Box Split 1 Split 2 Split 3 Chain to Fan Out Chain Fan In Fan Out	34
Triangle Split 1 Box Split 2 Split 3 Chain to Fan Out Chain Fan In Fan Out	35

Combination	Combination Number
Triangle Split 1 Split 2 Box Split 3 Chain to Fan Out Chain Fan In Fan Out	36
Triangle Split 1 Split 2 Split 3 Chain to Fan Out Box Chain Fan In Fan Out	37
Triangle Split 1 Split 2 Split 3 Chain to Fan Out Chain Box Fan In Fan Out	38
Triangle Split 1 Split 2 Split 3 Chain to Fan Out Chain Fan In Box Fan Out	39
Triangle Split 1 Split 2 Split 3 Chain to Fan Out Chain Fan In Fan Out Box	40
Chain to Fan Out Triangle Split 1 Split 2 Split 3 Box Chain Fan In Fan Out	41
Triangle Chain to Fan Out Split 1 Split 2 Split 3 Box Chain Fan In Fan Out	42
Triangle Split 1 Chain to Fan Out Split 2 Split 3 Box Chain Fan In Fan Out	43
Triangle Split 1 Split 2 Chain to Fan Out Split 3 Box Chain Fan In Fan Out	44
Triangle Split 1 Split 2 Split 3 Chain to Fan Out Box Chain Fan In Fan Out	45
Triangle Split 1 Split 2 Split 3 Box Chain Chain to Fan Out Fan In Fan Out	46
Triangle Split 1 Split 2 Split 3 Box Chain Fan In Chain to Fan Out Fan Out	47
Triangle Split 1 Split 2 Split 3 Box Chain Fan In Fan Out Chain to Fan Out	48
Chain Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Fan In Fan Out	49
Triangle Chain Split 1 Split 2 Split 3 Box Chain to Fan Out Fan In Fan Out	50
Triangle Split 1 Chain Split 2 Split 3 Box Chain to Fan Out Fan In Fan Out	51
Triangle Split 1 Split 2 Chain Split 3 Box Chain to Fan Out Fan In Fan Out	52
Triangle Split 1 Split 2 Split 3 Box Chain Chain to Fan Out Fan In Fan Out	53
Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan In Fan Out	54
Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Fan In Chain Fan Out	55
Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Fan In Fan Out Chain	56
Fan In Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan Out	57
Triangle Fan In Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan Out	58
Triangle Split 1 Fan In Split 2 Split 3 Box Chain to Fan Out Chain Fan Out	59
Triangle Split 1 Split 2 Fan In Split 3 Box Chain to Fan Out Chain Fan Out	60
Triangle Split 1 Split 2 Split 3 Fan In Box Chain to Fan Out Chain Fan Out	61
Triangle Split 1 Split 2 Split 3 Box Fan In Chain to Fan Out Chain Fan Out	62
Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Fan In Chain Fan Out	63
Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan Out Fan In	64
Fan Out Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan In	65
Triangle Fan Out Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan In	66
Triangle Split 1 Fan Out Split 2 Split 3 Box Chain to Fan Out Chain Fan In	67
Triangle Split 1 Split 2 Fan Out Split 3 Box Chain to Fan Out Chain Fan In	68
Triangle Split 1 Split 2 Split 3 Fan Out Box Chain to Fan Out Chain Fan In	69
Triangle Split 1 Split 2 Split 3 Box Fan Out Chain to Fan Out Chain Fan In	70
Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Fan Out Chain Fan In	71
Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan Out Fan In	72

Table 16 - The orders of match sets that were tried

Figure 47 shows how many valid matches it is possible to produce by using each method in each graph. The orders random, descending whole, descending individual and every other whole created the maximum valid matches to the common model graphs in call graph of Combine2-1. Random sorting the file and processing it, will neither produce the best result every time. In this graph it produced the best results after sorting the file 60 times, it may be the first in the next. In future analysis this order is ignored. However sorting the matches into orders descending whole and descending individual did not produce the best results in the call graphs of c-decl2 and real-2. There is therefore no clear method of maximising the number of valid matches that are produced

by the Match Analyser. If it is the intention to maximise the number of common model graphs in the eventual layout then this experiment shows that a 'first come first serve' method of match analysis is not viable. More likely the definition of a valid match is restrictive. This is because the current implementation of the Match Analyser reduces the matches found by over 97 percent. Often taking a graph with 600 matches to the common model graphs to less than 10 that are valid.

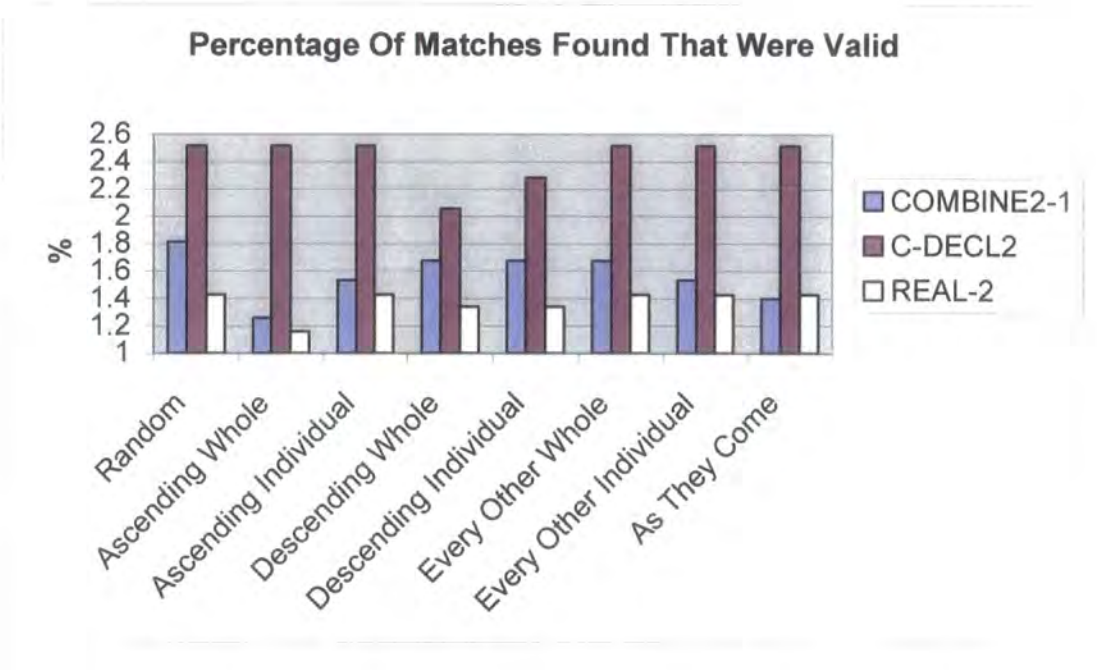


Figure 47 - The percentage of valid matches were possible by each order

The value used in Figure 47 is the average number of valid matches produced by the combinations. Further analysis of the fact base of valid matches that is produced by the Match Analyser is performed. It is performed to ascertain which combination of sets produces the minimum number of valid matches. The results are shown in Figure 48, Figure 49 and Figure 50.

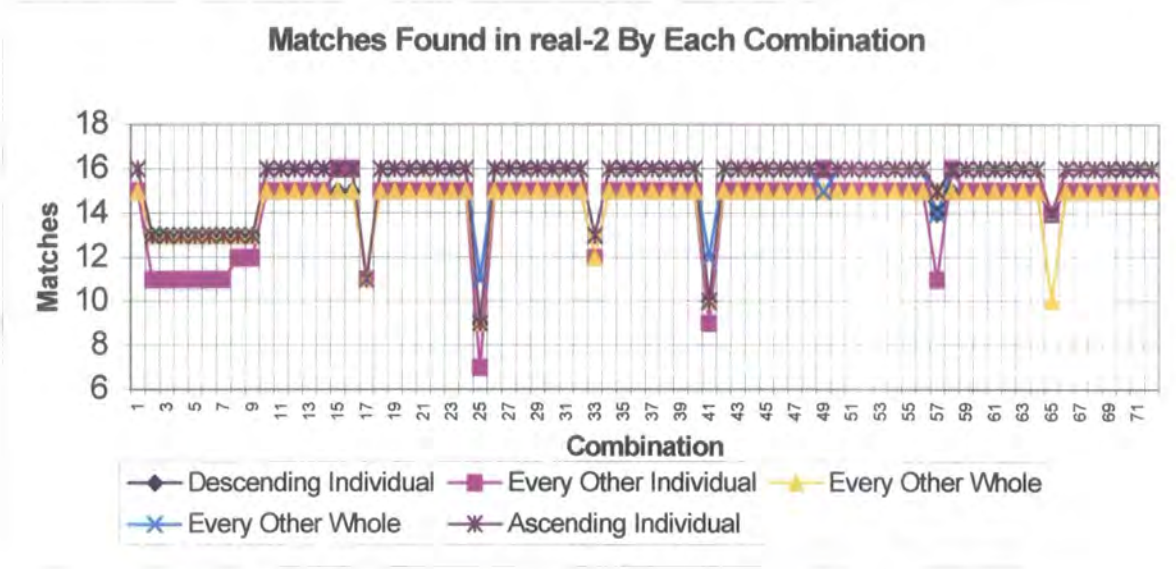


Figure 48 - The number of valid matches that each method produces in each combination in the call graph of real 2

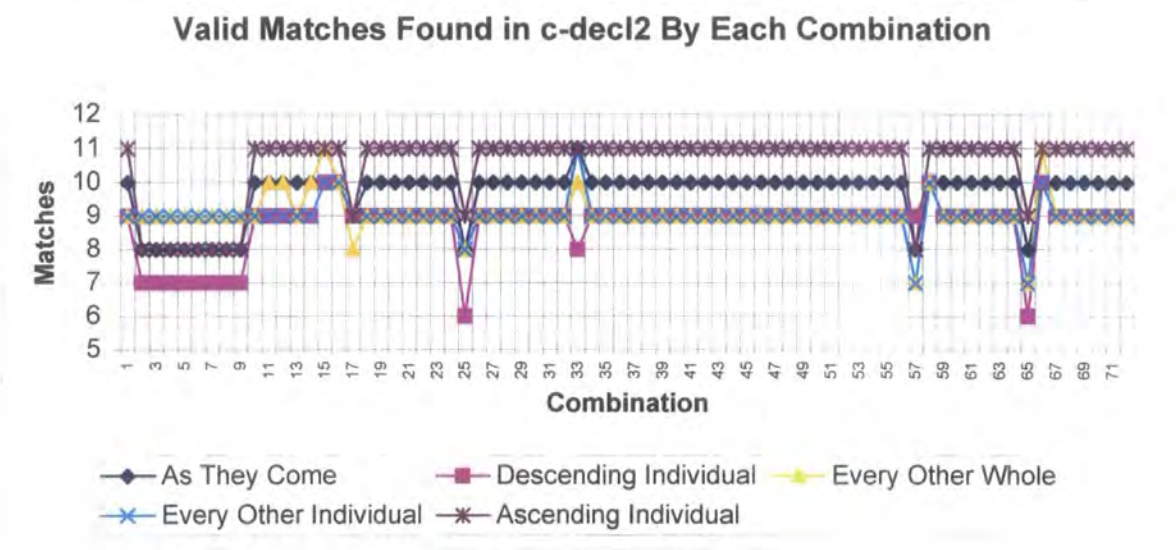


Figure 49 - The number of valid matches that each method produces in each combination in the call graph of c-decl2

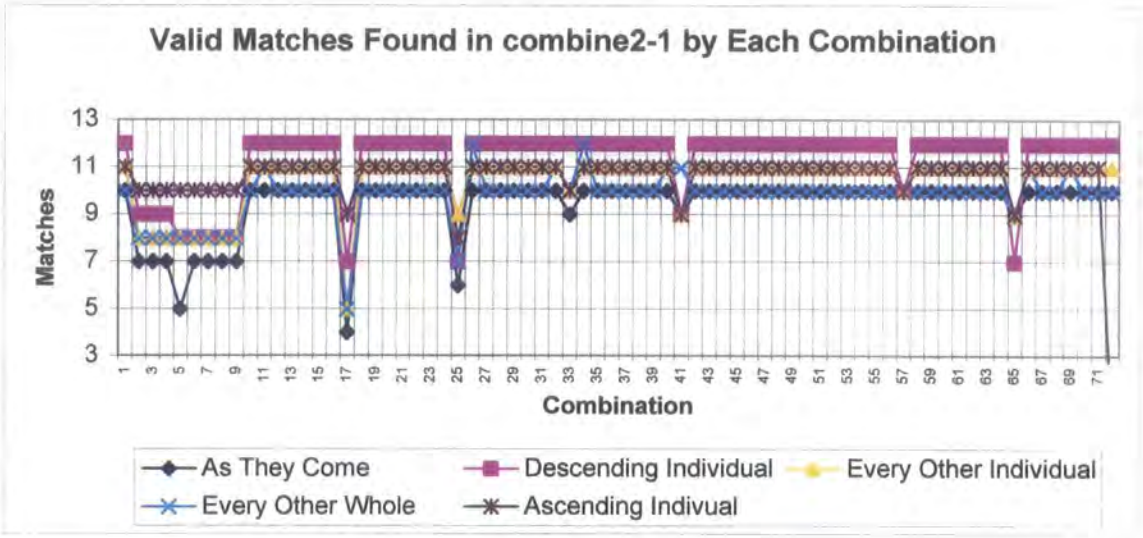


Figure 50 - The number of valid matches that each method produces in each combination in the call graph of combine2-1

It can be seen from Figure 48, Figure 49 and Figure 50 that if the goal is to maximise the number of valid matches then the combinations given in Table 17 should be avoided for every method. Further investigation proved that these combinations are those where the larger models, in terms of vertices involved and the split models are listed first. Therefore these take up a large number of the vertices that cannot be involved in any other common models and, therefore reduces the number of valid matches.

Combination	Combination Number
Split 1 Triangle Split 2 Split 3 Box Chain to Fan Out Chain Fan In Fan Out	2
Split 1 Split 2 Triangle Split 3 Box Chain to Fan Out Chain Fan In Fan Out	3
Split 1 Split 2 Split 3 Triangle Box Chain to Fan Out Chain Fan In Fan Out	4
Split 1 Split 2 Split 3 Box Triangle Chain to Fan Out Chain Fan In Fan Out	5
Split 1 Split 2 Split 3 Box Chain to Fan Out Triangle Chain Fan In Fan Out	6
Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Triangle Fan In Fan Out	7
Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan In Triangle Fan Out	8
Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan In Fan Out Triangle	9
Triangle Split 2 Split 1 Split 3 Box Chain to Fan Out Chain Fan In Fan Out	10
Split 2 Triangle Split 1 Split 3 Box Chain to Fan Out Chain Fan In Fan Out	17
Split 3 Triangle Split 1 Split 2 Box Chain to Fan Out Chain Fan In Fan Out	25
Triangle Split 1 Split 2 Box Chain to Fan Out Chain Fan In Fan Out Split 3	32
Box Triangle Split 1 Split 2 Split 3 Chain to Fan Out Chain Fan In Fan Out	33
Triangle Box Split 1 Split 2 Split 3 Chain to Fan Out Chain Fan In Fan Out	34
Chain to Fan Out Triangle Split 1 Split 2 Split 3 Box Chain Fan In Fan Out	41
Fan In Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan Out	57
Fan Out Triangle Split 1 Split 2 Split 3 Box Chain to Fan Out Chain Fan In	65

Table 17 - The combinations that do not maximise the number of valid matches

From the above it is not possible to ascertain which method or combination maximises the number of valid matches. It can however be narrowed down to three methods which produce the best results and can be easily created. They are those of taking alternate matches, sorting into ascending order and descending order. In Chapter 8, these three orders and the natural order are tested to see which method produces the best quality of graphs in terms of the metrics of the graphs.

6.2.2 Natural Order

In Chapter 4 it is seen that many models are made up of other models and primitive models. This causes several orders to appear which may allow more vertices to be involved in fewer models. However the orders all place the larger models (Split 1, Split 2 and Split 3) and therefore the orders are such that it involves as many vertices in as few models as possible.

The call graph of the program 'recog' (Table 12 shows the properties of the graph) is analysed for matches to the common model graphs manually. A list of valid matches produced by following **AnalysisOfMatches** (**Algorithm 3**) in Chapter 4 are used to layout the graph manually. The matches are sent through the algorithm in three combinations. The matches are combined in the order they are found and not sorted in anyway. The orders are given in Table 18. This provides experience of carrying out the process and giving a valuable insight into the problems that are to be faced when implementing the Graph Layout System. It also provides the opportunity to test the ANHOF method at an early stage of its development.

Combination	Combination Number
Split 2, Split 1, Split 3, Chain to Fan Out, Box, Triangle, Fan In, Fan Out, Chain	1
Split 3, Split 2, Split 1, Chain to Fan Out, Fan Out, Fan In Chain, Box, Triangle	2
Split 2, Split 3, Split 1, Chain to Fan Out, Chain, Triangle, Fan Out, Fan In, Box	3

Table 18 - The natural orders to send matches through the Match Analyser

Intuitively. order one produces the best looking graphs and the graphs are easy to layout because many of the vertices in the graph are in few common model graphs. Investigating this order produces a similar number of valid matches as combination 17 and therefore does not maximise the number of matches. In Chapter 8 it is shown that this produces the best looking graphs in terms of metrics of the graphs. Therefore if the intention is to maximise the metrics of the graph then vertices should be involved in the larger models rather than the smaller ones. To perform this it is suggested that order one is used to send the common model graph matches through the Match Analyser.

6.3 Graph Layout System

The Graph Layout System is used to layout the whole graph and the common model graph. In Chapter 4 various issues become apparent that need investigating in order to optimise the automatic graph layout algorithms given in Algorithm 5 to Algorithm 17. They need to be optimised so that the best layouts in terms of the metrics can be achieved. One issue is the order in which the vertices are placed on the plane. This is critical because the order in which the vertices are placed on the plane may prevent edge crossings. This order is investigated in the section below.

Another issue that is made apparent in Chapter 4 is the aesthetics properties that should be applied to the graph. The ANHOF system is a prototype system causing few aesthetic properties to be implemented. These are the spaces between vertices on the horizontal and vertical axes. The settings for these are also given below.

6.3.1 Vertex Order

It is seen later in Chapter 8 that using the algorithm used in Graph Tool as the 'Standard' algorithm (step 8 in **LayoutRepresentation (Algorithm 4)**) in the ANHOF method produces the best results. This is given in **GraphLayout** (Algorithm 17). As part of this algorithm and the automatic graph layout algorithms for the models it is necessary to sort the vertices into order. When a tree is laid out the order in which the children are placed is critical in terms of reducing the edge crossings and the comprehension. In the algorithms in Chapter 4 the order describes how they are placed

on the plane. Given below are some of the options for sorting the vertices. In the diagrams below the vertices are labelled with the procedure name in capitals and the fan in and fan out values as a pair below (fan out, fan in).

A simple nine vertex graph G is used to illustrate the orders used. This is given below: -

G=(V,E)

V={A, B , C, D, E, F, G, H, I}
E={(A,B), (A,C), (B,C), (B,D), (B,F), (B,E), (C,D), (E,G), (E,H), (F,G), (D,G), (G,I)}

6.3.1.1 Descending Alphabetical Order

This is the method used in Graph Tool to sort the vertices. It is reasonably successful, both because of its simplicity and its reduction of edge crossings. It is more successful on small graphs, in the layout of graph G given in Figure 51, the method causes zero crossings and hence, is one of the best.

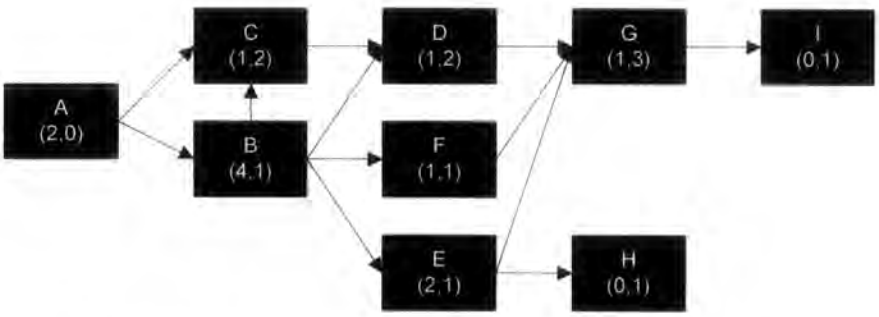


Figure 51- The descending alphabetical ordering of vertices

6.3.1.2 Ascending Alphabetical Order

Sorting the vertices into ascending alphabetical order can be successful on small graphs. However in the example it caused two edge crossings and a layout that occupies more area because the maximum number of vertices on each level is greater than the other layouts. In practice it works as well as sorting them into descending alphabetical order.

When graph G is laid out using this sort order it is laid out in the manner given in Figure 52.

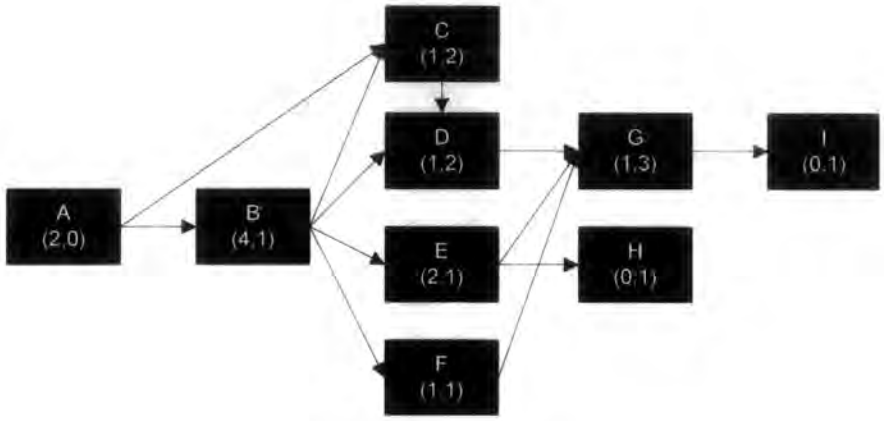


Figure 52 - The ascending alphabetical ordering of vertices

6.3.1.3 Fan In Ascending

If the vertices of graph G are sorted by their fan in properties, so that the lowest fan in value is at the top, a diagram similar to that in Figure 53 is produced. The graph is quite high and there is one crossing.

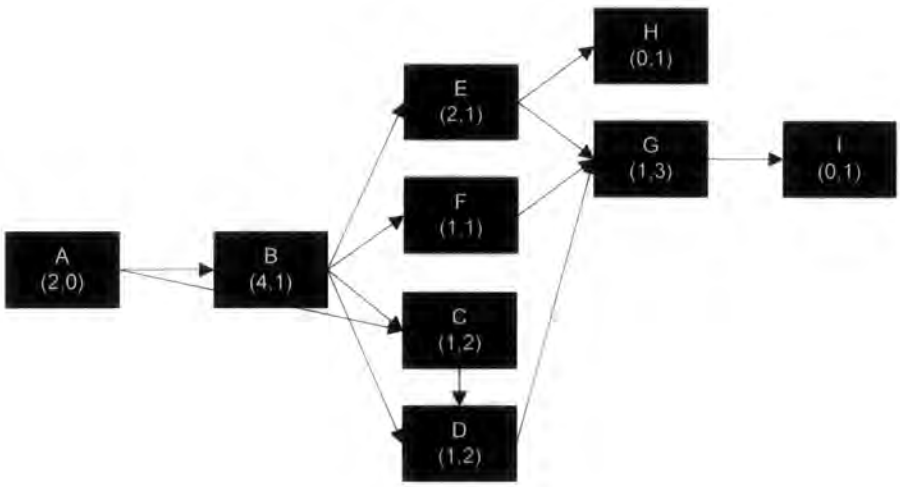


Figure 53 – The fan in ascending ordering of vertices

6.3.1.4 Fan In Descending

Figure 54 shows how graph G would be laid out if the vertices were sorted into fan in value descending order. So that the highest fan in is at the top of the plane. It is a similar graph layout to that of Figure 51, except there is one crossing, caused by the vertex labelled 'E' being placed above F. This is because the vertices are always loaded up in identification number order, E is given an identification number that is less than the number for vertex F causing the layout below. This shows that there are other underlying vertex orders imposed to solve order problems that occur. These problems often occur when vertices have the same fan in value or comparison value.

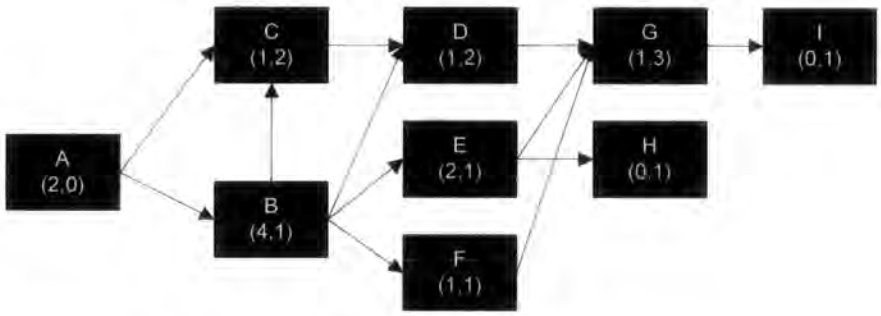


Figure 54 – The fan in descending ordering of vertices

6.3.1.5 Fan Out Ascending

Sorting the vertices of graph G into fan out ascending order produces the same diagram as Figure 51 that has zero crossings. In practice it produces better results than sorting alphabetically because the vertices are sorted by graph specific properties which when laying out is more reliable than a human imposed label. It was found in trials that you could sort the vertices according to their degree but it was common for vertices to have the same degree, therefore it was found to be better to break the degree figure into the fan in and fan out properties, and sort on them. Vertices may still have the same fan out or fan in property.

6.3.1.6 Fan Out Descending

Sorting the vertices of graph G into fan out descending order produces the same diagram as Figure 52 that has three crossings. In practice this works as well as sorting into fan out ascending. It works on different styles of graphs.

6.3.1.7 Topological Sort

A topological sort [170] is a standard graph vertex-sorting algorithm. It is a method of making sure that if there is a path from v_i to v_j then v_j appears after v_i in the ordering. The algorithm is based on a queue of the fan information. Of course the method then depends on the order the vertices are placed on the queue. If the vertices in graph G are placed so that vertex B is placed after vertices E and F then a layout is achieved like that in Figure 53 and therefore no crossings are caused. Whereas, if vertex B is placed in some variation of vertices E, F, and B then no crossings are caused but the edge between vertices and D may be hidden from view. In tests this method of sorting was found to be more difficult to implement than the other methods, and generally produces inferior laid out graphs in terms of hidden edges and it tended to increase the edge crossings.

6.3.1.8 Combination of Orders

It can be seen above that there is no ideal way that will work on every size and type of graph. It is found by applying these orders to the graphs in Chapters 7 and 8 that it is best to impose a combination of all three orders and therefore avoiding the problem of the tool imposed vertex identification number being the final sorting factor. In certain circumstances sorting on the fan in value is best and in others sorting on the fan out value is best. After many trials it was found that vertices have a more varying fan out value than fan in value. It makes little difference if they are sorted into ascending or descending order. Therefore the order that is used when sorting vertices in the ANHOF method is to sort the vertices into descending fan out order then descending fan in order for any vertices with the same fan out order, finally for any vertices that have the same fan in and fan out value sort them in descending alphabetical order. In the case of graph

G this produces a layout that has three crossings (shown in Figure 55) but in practice reduces the crossings in larger real world graphs.

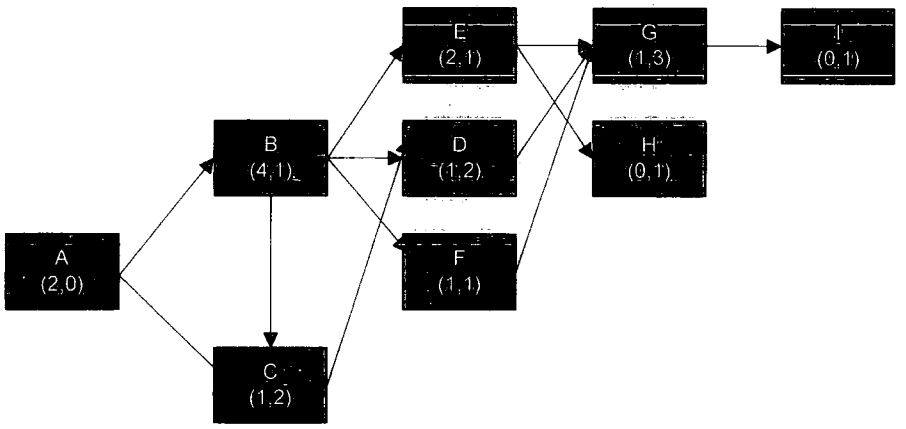


Figure 55 - The vertices sorted in a combination of orders

6.3.1.9 Heuristics

Sugiyama et al. [159] suggests several heuristics for sorting the vertices on a level. From this paper 'bary centre' ordering provides a method of sorting them that is simple to implement. These methods may present an improvement over the above method. Another heuristic that may be an improvement on the above is to sort the vertices so that the ones with lowest vertex degree levels are in the middle and the ones with the highest vertex degrees are on the outside. But the highest degree vertices are placed either on the top or bottom of the list of children, in the position that minimizes the edge length of the vertices in the graph.

6.3.2 Aesthetic Settings

It is suggested in Chapter 2 that it is desirable to minimize the area taken by the graph and also to produce a ratio between the sides that is close to 1.41. In order to achieve this the spacing on vertices should be set. The vertical spacing should be less than the horizontal spacing in order to achieve the ratio. Trials of graphs using the Graph Tool display system shows that the horizontal (X axis) spacing should be set 100 and vertical (Y axis) spacing should be set 50. This enables all arrowheads to be seen and for edges to be of reasonable length.

6.4 Summary

In this chapter the settings are given to maximise the performance of the ANHOF system in terms of improving the metrics of the eventual layouts and aiding comprehension. The chapter shows the settings for the length of chains and the levels of fan in and fan out properties of vertices in order to get the maximum number of matches to the various variable models using the Variable Model Detection system. It also shows that the implementation of Ullman's isomorphism algorithm in Messmer's [114] Graph Matching Toolkit is the best isomorphism algorithm to use for the Fixed Isomorphism Detection system. These settings are necessary to maximise the number of matches produced by the Graph Isomorphism System.

The chapter details an order to combine the set of matches to the various common model graphs to cause the maximum number of models to pass through the Match Analyser and become valid matches. This may be used to increase the comprehension of the graph by having the maximum number of common model graphs present in the eventual layout. After further investigation in Chapter 8 it may prove that maximising the number of valid common model graph matches may improve the metrics of the graph and it gives a natural order to combine the set of matches to the various common model graphs that may also achieve the best metrics for the eventual layout of the graph when passed through the Match Analyser.

The chapter also suggests an order in which the vertices be sorted which will be used to obtain the next available vertex in the automatic graph layout algorithms for the common model graphs. The sort order of the vertices is discussed so that the crossings are minimized. This sort order is important because it may reduce the crossings further and is necessary so that the Graph Layout System can be optimised. Finally the spacing of the vertices is given so that the ratio between the horizontal and vertical axes is minimized.

In the next chapter the layouts obtained from the ANHOF system are compared with other layouts form Graph Tool and daVinci.

7. The ANHOF System at Work

Chapter 4 gives a description of the ANHOF method. This is then implemented using the ANHOF system discussed in Chapter 5. Chapter 6 provides the settings necessary to optimise the ANHOF system. In this chapter the layouts created by the ANHOF system using these settings are compared with those generated by existing layouts algorithms. It shows some of the problems associated with conventional layouts and how they are corrected using the ANHOF System. First of all, a simple example is given, and then four call graphs from the GCC version 2.58 are laid out and compared before the graphs layouts are compared using metrics.

7.1 Simple Example

In order to show how each part of the ANHOF system works an example graph is constructed, simple enough to show every feature of the various parts. It consists of 45 vertices and 50 edges. A possible layout is shown in Figure 57 of Graph G defined below.

Given a graph G=(V,E)

Where : -

V= {1,2,4,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25, 26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45}

E= {(1,18), (1,2), (1,37), (1,41), (1,6), (2,3), (3,4), (4,5), (6,7), (7,10), (8,10), (9,10), (10,11), (10,12), (10,13), (14,15), (15,17), (16,14), (17,16), (18,19), (19,20), (20,21), (20,22), (20,23), (21,14), (22,24), (23,32), (24,25), (24,26), (24,27), (28,25), (28,26), (28,27), (28,29), (28,30), (28,31), (32,33), (32,34), (32,35), (36,33), (36,34), (36,35), (37,38), (38,39), (38,40), (40,39), (41,42), (43,42), (44,42), (45,42)}

7.1.1 Existing Automatic Graph Layout Algorithms

Two existing layout algorithms are compared in the following chapter. The tools named Graph Tool and daVinci implement specific layout algorithms, and are used to layout the graph; the resulting layouts are given below and discussed.

7.1.1.1 Graph Tool

Graph Tool is a graph display tool originally developed in Durham by Bodhuin [16] and later maintained by Young [180]. It is possible to layout graph G using Graph Tool. The output is shown in Figure 57. Graph Tool adopts a straight-line standard of graph layout where each edge is a straight line, forming the shortest route between two vertices. This has many problems that can be summarised as: -

- related vertices are not situated with each other,
- common Structures are not apparent,
- come edges are lost, and
- high edge crossings.

The vertices that are connected to each other should be as close together as possible. For instance vertices 8, 9, 10, 11, 12 and 13 should be close to 7 as should vertices 42, 43, 44 and 45 be close to vertex 41. Vertices 14, 15, 16, 17 should be close to vertex 21 and 25, 26, 27, 28 should be close to vertex 24. There may be other instances where this is the case but this is not clear from the diagram. The common models graphs in the ANHOF method can be used to force this closeness.

The common structures discussed in Chapter 4 are not apparent in the layout performed by this graph layout tool. Many of the models laid out down the edge of the diagram cause the hierarchical nature of the graph (all vertices leading from vertices labelled 1) to be lost. But some of the models can be seen. Some models are clear but are not laid out as they would be using their associated automatic graph layout algorithms. For instance the Box model of the vertices labelled 16, 17, 14, 15 should be laid out like a Box not a form of rhombus.

In the diagram some edges are not shown. They merge themselves with other edges. For instance the Triangle structure between vertices 38, 39, and 40 is shown as a chain of vertices like in Figure 56(a) but however there is an edge between vertices 38 and 39. Therefore it should be laid out as a Triangle shown in Figure 56(b)

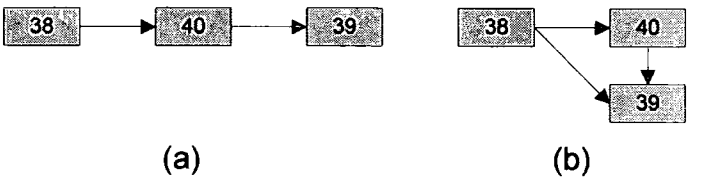


Figure 56 - The Triangle structure (a) laid out using Graph Tool (b) correctly laid out

It is desirable not to have edge crossings in a diagram because they make it easier to follow. In this diagram there are 13 line crossings, this is not a considerable number but given that there are 50 edges, this is a fair proportion. Most of the edge crossings could be avoided if the related vertices were close together.

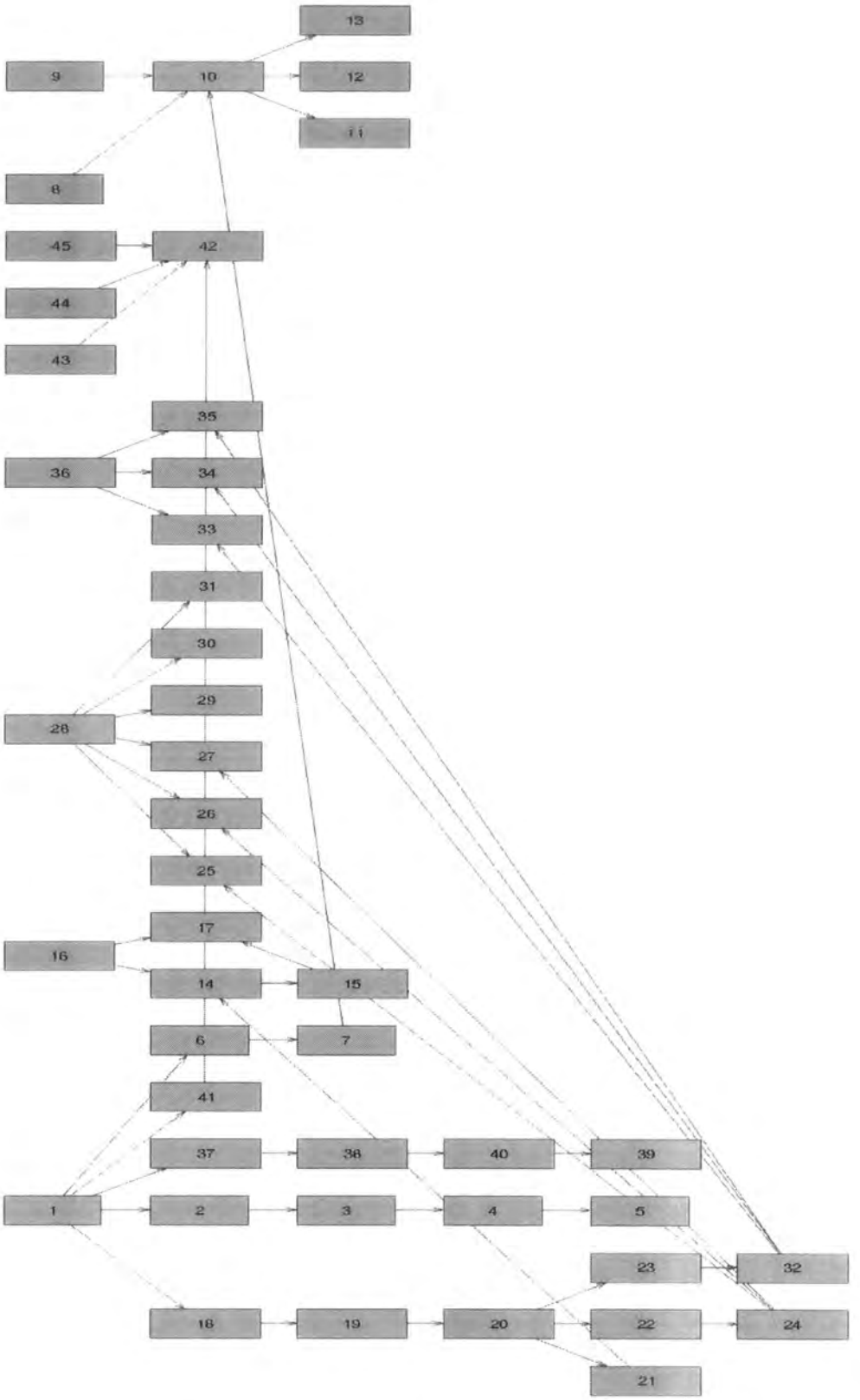


Figure 57 - How graph G is laid out using Graph Tool

7.1.1.2 daVinci

The University of Bern developed a powerful graph drawing tool that uses the algorithm of Sugiyama et al. [159] to layout all graphs. This tool is called daVinci and can be seen in [64]. Graph G can be laid out using the daVinci System as shown in Figure 58. One improvement of the daVinci layout of Graph G over the other tools is that it improved the ratio between the height and width of the graph. In Chapter 8 it is shown that European paper formats have a ratio of 1.41 or a screen has a ratio of 1.33, these figures are to be aimed for. In this diagram it is closer to 1 than any of the other layouts and is therefore the layout to achieve this metric. However this is not the case in practice where the layouts are long and thin giving a high ratio. The Graph Layout System has similar problems to that of the Graph Tool mainly: -

- the related vertices are not situated with each other,
- the common structures are not apparent,
- high number edge crossings, and
- the hierarchy is difficult to follow.

Again daVinci does not position related vertices together, however it is different vertices to the diagram produced by Graph Tool. For instance 36 should be closer to vertices 34, 35, and 33. Also vertices 28, 31, 30, and 32 should be closer to 27,26, and 25.

There are fewer common structures shown in this diagram than in the Graph Tool Layout. It is closely related to the hierarchy created. A Triangle model is shown on three levels instead of two, for instance the Triangle between 38, 39 and 40. Many of the common structures are laid out differently to Chapter 4. This is because the algorithm does not centre father vertices above its children. For instance the central vertex labelled 10 of the Split3 model involving vertices labelled 7, 8, 9, 10, 11, 12, and 13 is not central therefore the model is not easily followed or detected. The Box structure of the vertices 14, 15, 16 and 17 is again hindered by the rigid hierarchical structure imposed on the diagram by the automatic graph layout algorithm; there is no need for these to span four levels.

The hierarchical structure imposes many line crossings on the diagram. This could be reduced with better sorting of vertices. For instance if vertex 28 was situated between vertices 24 and 14 with vertices 31, 30 and 29 situated in between 25 and 15 then many crossings could be avoided. This algorithm always creates more crossings than Graph Tool, possibly because Graph Tool has a better sorting algorithm.

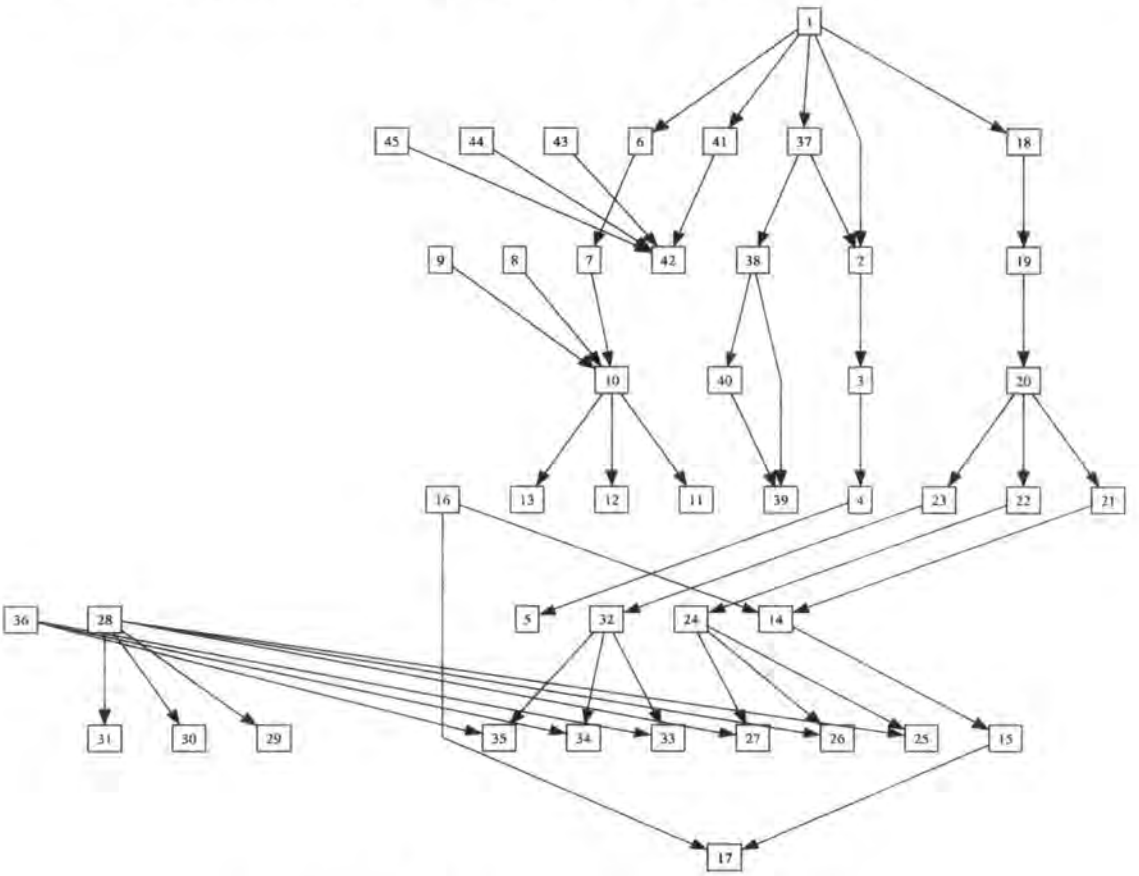


Figure 58 - How graph G is laid out using daVinci

7.1.2 The ANHOF System

The aim of the use of graph G is to illustrate how each part of the ANHOF system complimented each other improving the layout of call graphs. Using the models described in Chapter 4, graph G is fed through the ANHOF system. The input and output of each program of the ANHOF system is detailed in Appendix 4. The resulting, improved graph is shown in Figure 59. The graph is an improvement because of the following: -

- no edge crossings,

- common structures are clear,
- a good ratio between the width and height of the graph,
- the total edge length is reduced, and
- related vertices are situated together,

The detection of the models was correct, however the Triangle model involving vertices 38, 39 and 40 was not laid out using the Triangle automatic graph layout algorithm, it is laid out using a Chain to Fan Out model automatic graph layout algorithm. Which may be a case for changing the levels of the various settings of the detection algorithm or the order the Match Analyser processes them.

The Graph Display System that is used was Graph Tool. However, until recently, this did not allow edges to be connected to specific parts of a vertex. This caused edges to cross vertices that could have otherwise been avoided. For instance the edge connecting vertices labelled 1 and 41 crosses vertex 45, it could have been avoided if it was connected to the left hand edge of vertex labelled 41.

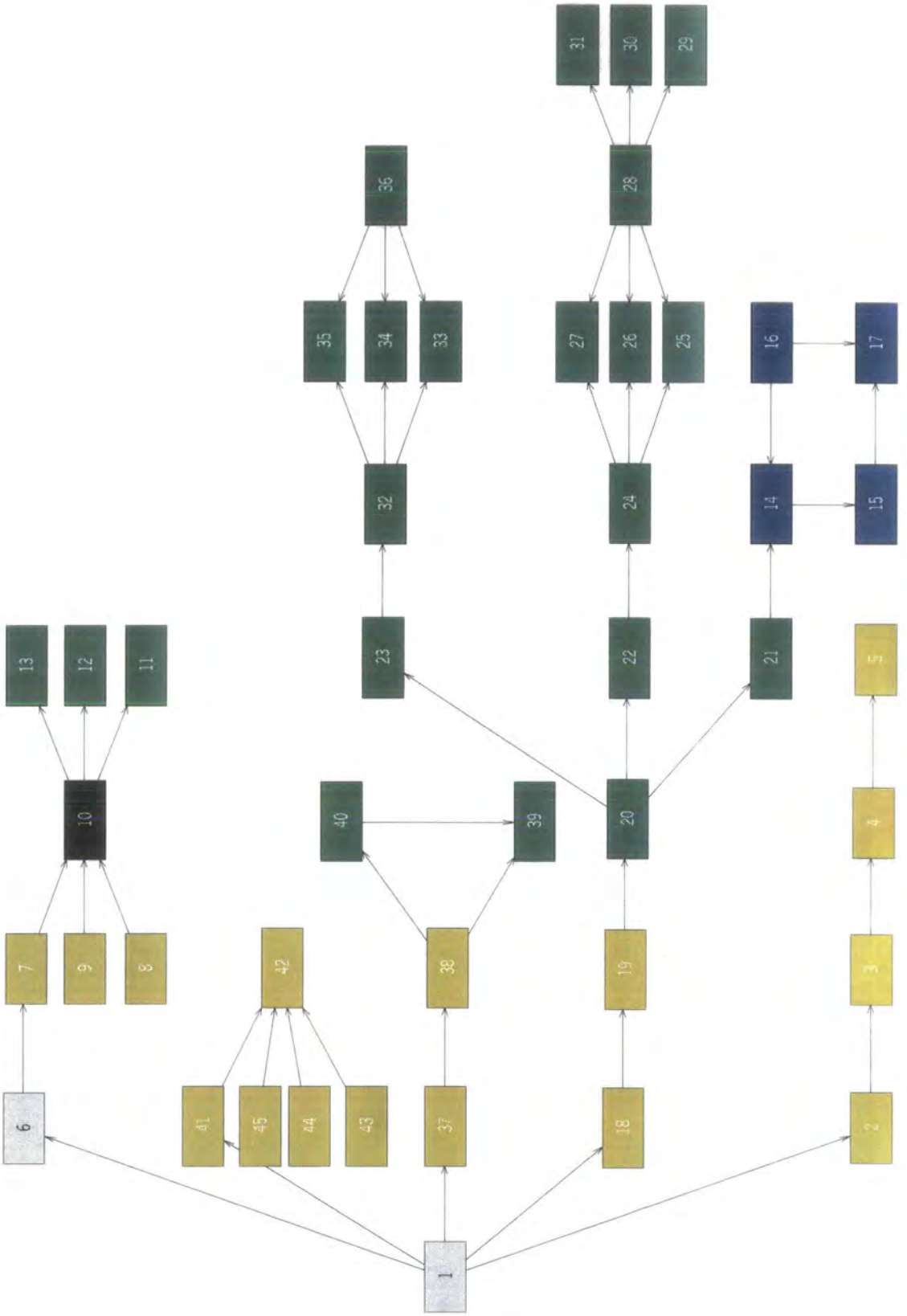


Figure 59 - How graph G is laid out using the ANHOF system

7.2 Real Call Graph Examples

The aim of the ANHOF method / system is to improve the layout of real world call graphs. Therefore in order to show the ANHOF system working, real call graphs from implemented program are used. The following examples are taken from GNU C compiler GCC version 2.58. There are many methods of extracting the call graph from the source code [118] or even the executable [52]. The call graphs used below are abstracted from the source code using a source code analyser for C programs know as CCG [94]. Four call graphs are shown from the programs of cp-search, genopint, varasm and localalloc.

7.2.1 Call Graph of cp-search

The call graph representing the program of cp-search is a nine vertex, eight edge graph that represents the procedures for searching GCC's internal representation of a C program. It is a very simple graph and again illustrates the ANHOF system working and highlights similarities between the Graph Tool automatic graph layout algorithm and the daVinci one.

7.2.1.1 Existing Automatic Graph Layout Algorithms

The layout obtained from Graph Tool is shown in Figure 60 and the layout obtained from daVinci is shown in Figure 61. They both fail to deal with the following properties of the graph: -

- the graph is not semantically recognized, and
- the Triangle Model is not visible.

Both of the automatic graph layout algorithms fail to highlight the Triangle model, involving vertices labelled 'pop_type_level', 'pop_stack_level' and 'obstack_free'. Graph tool misses out an edge as discussed earlier and the hierarchical structure imposed on the graph by daVinci makes it less obvious to recognize.

When a graph is not connected and it can be seen in the layout then it is said to be symmetrically recognisable. Both of the automatic graph layout algorithms layout the graph so that it looks to be one graph when it is two, and are therefore not semantically recognisable. In Graph Tool the edge between vertices labelled 'my_tree_cons' and '_obstack_newchunk' makes it seems that vertex labelled 'my_tree_cons' is part of the structure above it involving vertices labelled 'pop_type_level', 'pop_stack_level' and 'obstack_free' when this structure is a separate graph. This will hamper understanding of the graph. In the daVinci layout the edge between 'pop_stack_level' and 'obstack_free' makes it seem that the Triangle model between 'pop_type_level', 'pop_stack_level' and 'obstack_free' is part of the main graph when it is a separate graph.

The daVinci layout again causes more edge crossings than the other automatic graph layout algorithms. It is possibly because of the sorting algorithm. Swapping '_ostack_newchunk' with 'obstack_free', and 'my_tree_cons' was inserted in between 'pop_stack_level' and 'push_stack_level', would remove the edge crossings and also make the two graph structure visible.

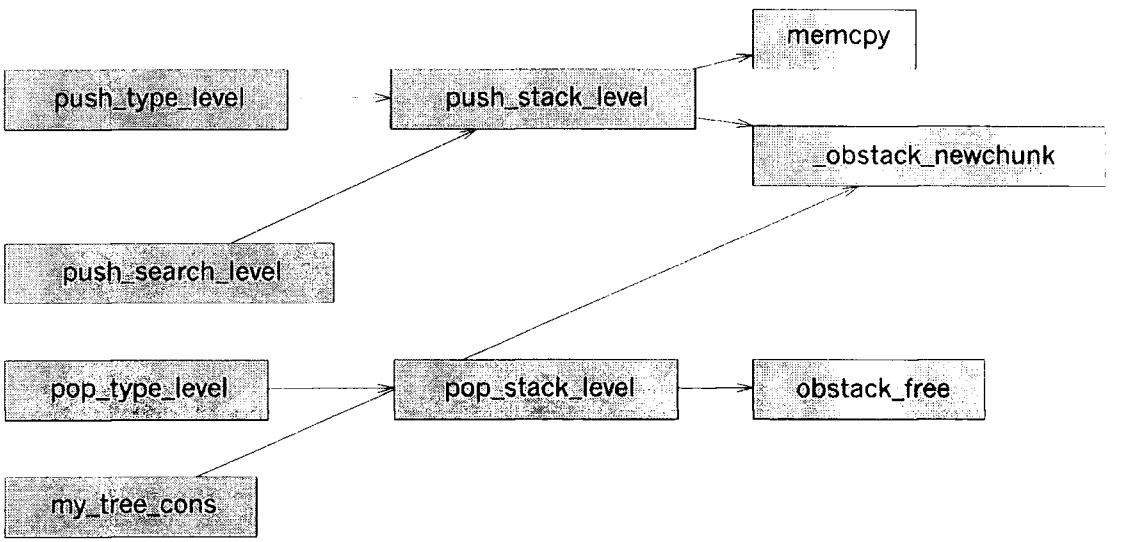


Figure 60 - The layout of program cp-search using Graph Tool

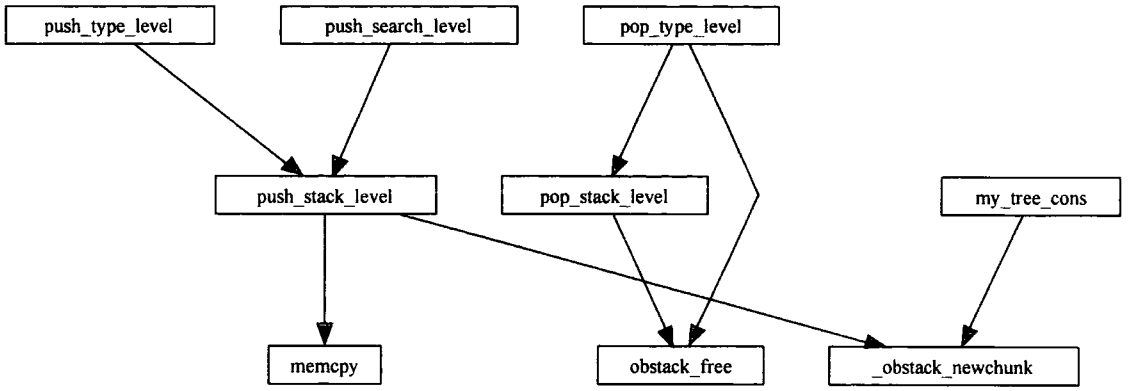


Figure 61 - The layout of program cp-search using daVinci

7.2.1.2 ANHOF System

The layout obtained by sending the call graph of the program cp-search through the ANHOF system is shown in Figure 62. The two-graph structure can be quite clearly seen as can the models present. In Figure 62 an edge crosses a vertex, the use of the straightline standard of edge routing has caused this. The edge is between vertices, labelled 'my_tree_cons' and '_obstract_newchunk', it crosses the vertex labelled 'push_search_level'. This is caused by step 8 of the automatic graph layout algorithm (**LayoutRepresentation** (Algorithm 4)) laying out the whole graph as three vertices, one for 'my_tree_cons', one for the Split 3 model (vertices labelled 'push_search_level', 'push_type_level', 'push_stack_level', '_obstract_newchunk' and 'memcpy') and another for the Triangle model (vertices labelled 'pop_type_level', 'pop_stack_level' and 'obstack_free').

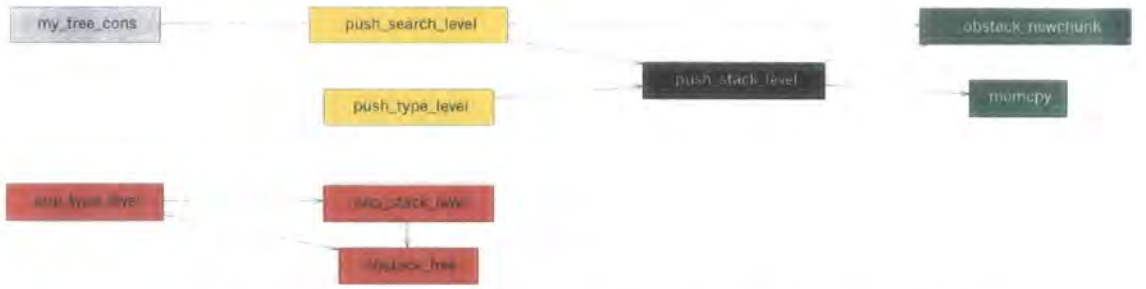


Figure 62 - The layout of program cp - search using the ANHOF System

7.2.2 Call Graph of genopinit

Many modules in the GCC compiler need to process an array of operation codes. The function to process these operation codes are in the module called genopinit. Given below is a discussion of the call graph of the module laid out by various methods. The call graph consists of 21 vertices and 22 edges.

7.2.2.1 Existing Automatic Graph Layout Algorithms

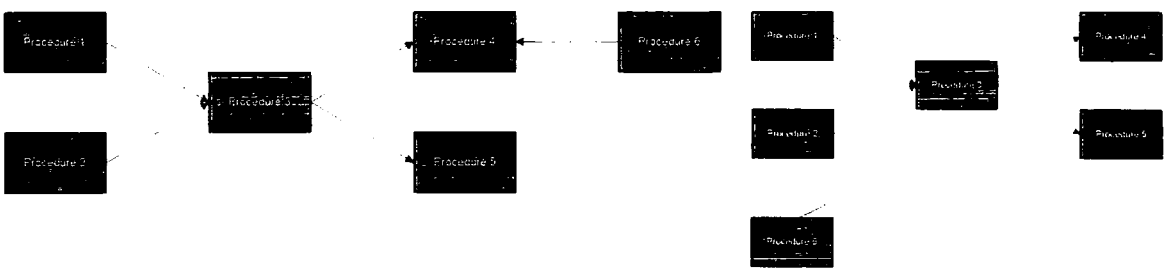
The layout obtained by using daVinci to layout the call graph of genopinit is shown in Figure 64(a) and the layout using Graph Tool is shown in Figure 64(b). This graph is larger and so the problems of conventional automatic graph layout algorithms are becoming evident and are: -

- related vertices are not close together,
- edges are crossing unnecessarily,
- hard to follow, and
- common structures are not apparent.

The daVinci layout is increasingly becoming harder to follow. Its vertex-sorting algorithm separates related vertices and elongates edges more than necessary. For instance, the routing of the edge between vertices labelled 'exit' and 'main' causes the edge to be longer than necessary. If the vertex labelled 'exit' is positioned on the opposite of the vertex labelled 'fprintf' then the edge could have been routed along the

other side of the graph, towards the vertex labelled 'fatal'. It may not be much shorter but many edge crossings could be avoided. Similarly if the vertices labelled 'xrealalloc' and 'xmalloc' are positioned on the other side of the vertex labelled 'main' many more edge crossings are avoided.

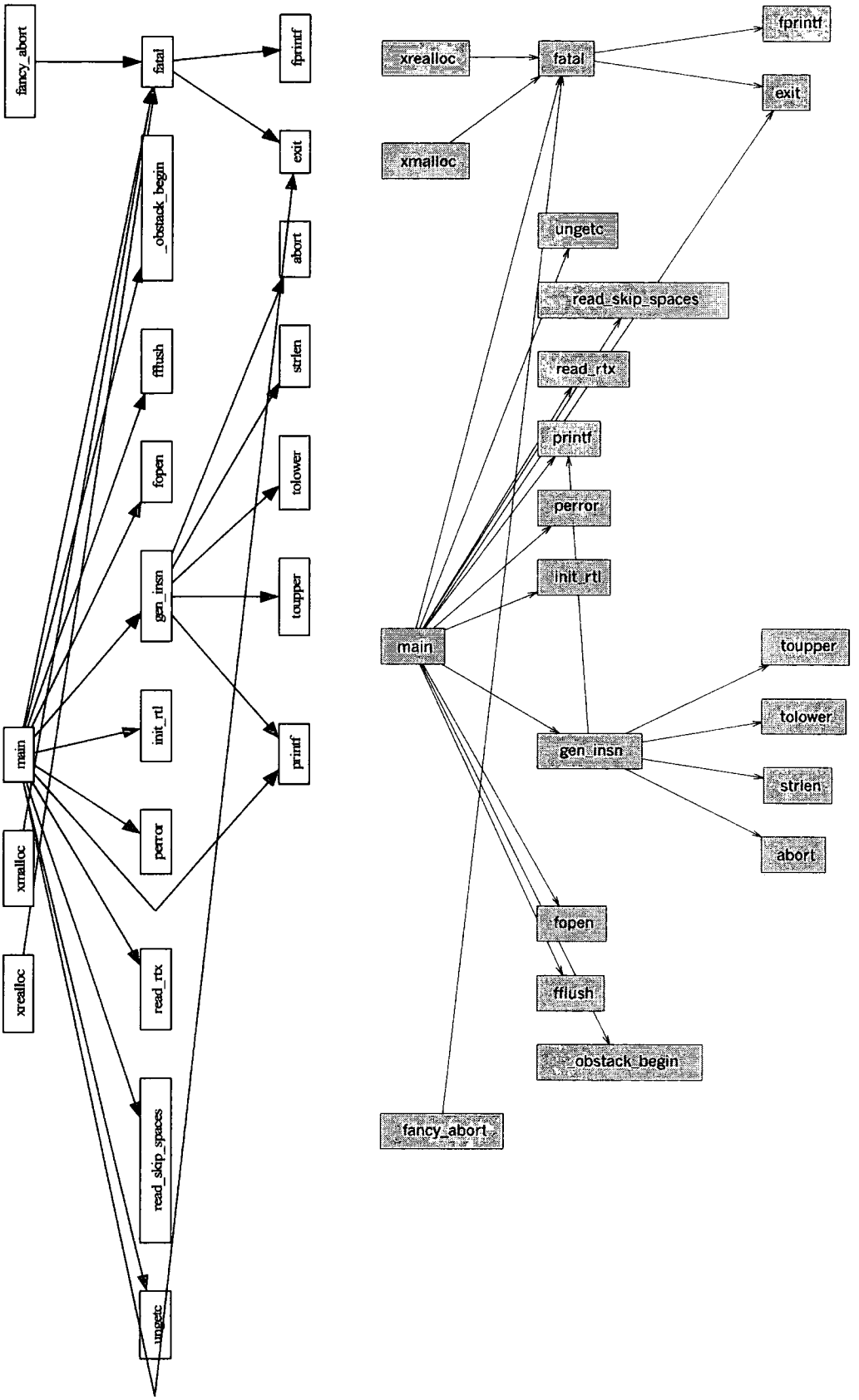
In Graph Tool the vertices are sorted alphabetically and so the vertices labelled 'xrealalloc' and 'xmalloc' are on the right side of main. However the vertex labelled 'fancy_abort' is the wrong side of 'main'. This is a simple example of a problem with other graphs in that the automatic graph layout algorithm in Graph Tool cannot cope with a single vertex flowing into another vertex that has already been placed. If this was implemented the Graph Tool automatic graph layout algorithm would perform much better. As an example the graph shown in Figure 63(a) would be laid out like the graph in Figure 63(b). This is because the automatic graph layout algorithm in Graph Tool starts at one vertex and flows to the next and continues until no more vertices flow from the current vertex. The algorithm should check that there are no more vertices that flow into the current vertex when there are no more vertices that flow out.



(a) Improved Layout **(b)Graph Tool Layout**

Figure 63 - An example graph and how its layout could be improved

The common structures that are present in the graph are more obvious using the Graph Tool Layout than they are using the daVinci automatic graph layout algorithm. but this is largely because it is easier to read the graph when the writing is flowing across the page (across the shortest side of the page) than the automatic graph layout algorithm. Again daVinci causes more edge crossings than the other two automatic graph layout algorithms.



(a) By daVinci (b) By Graph Tool

Figure 64 - genopinit laid out using conventional layout tools

7.2.2.2 The ANHOF System

If the models given in Chapter 4 are used to pass the call graph from genopinit program through the ANHOF system produces a graph laid out in the manner given in Figure 65. Here the common structures are easily seen and the related vertices are situated together. However the sorting of the vertices that flow out from the vertex labelled 'gen_insn' causes edge crossings. The edge between vertices labelled 'main' and 'printf' crosses some of the edges that flow out from the vertex labelled 'gen_insn'. This could be avoided if the vertices were sorted differently or a better heuristic was applied, such as those discussed in Chapter 6.

The problem that vertices cannot be a member of two or more models is highlighted here. The vertices that flow out from the vertex labelled 'main' form a Fan Out model. However vertices labelled 'gen_insn' and 'fatal' are members of other models that are given preference to the Fan Out common model graph involving the vertex labelled 'main'. It is fortunate that the 'standard' automatic graph layout algorithm can cope with the Fan Out model naturally. It may be best to take the vertex that is in two or more models out of the offending model match, so that most of that model is laid out using a common model algorithm. However this would not make any difference to the layout of this graph.

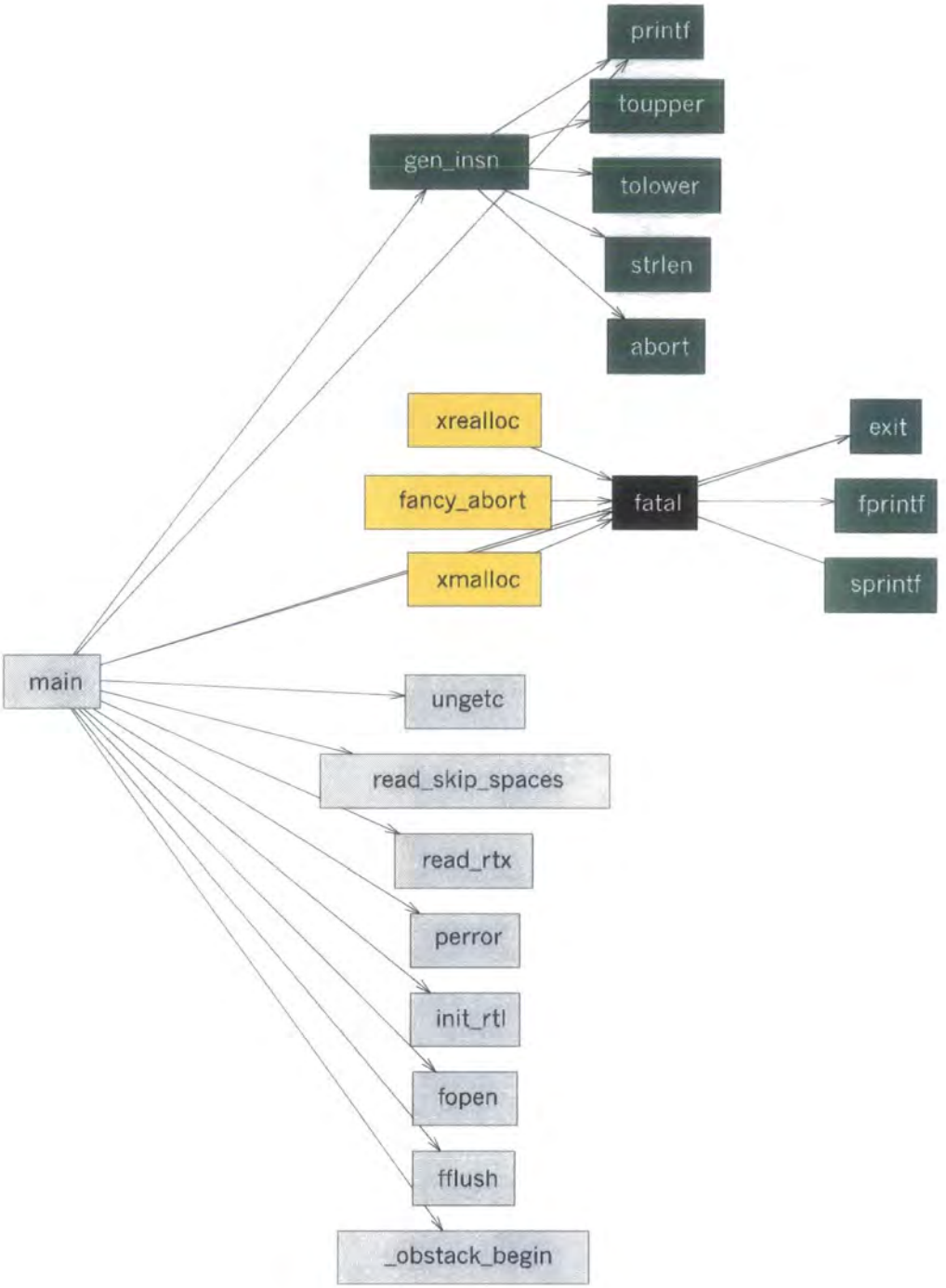


Figure 65- genopinit laid out using the ANHOF system

7.2.3 Call Graph of varasm

The call graph of the program varasm compiles the assembler of a C program. It can be represented as a call graph with 26 vertices and 32 edges. Detailed below is a

comparison of the layout obtained from using the automatic graph layout algorithms of daVinci, Graph Tool and the ANHOF system.

7.2.3.1 Existing Automatic Graph Layout Algorithms

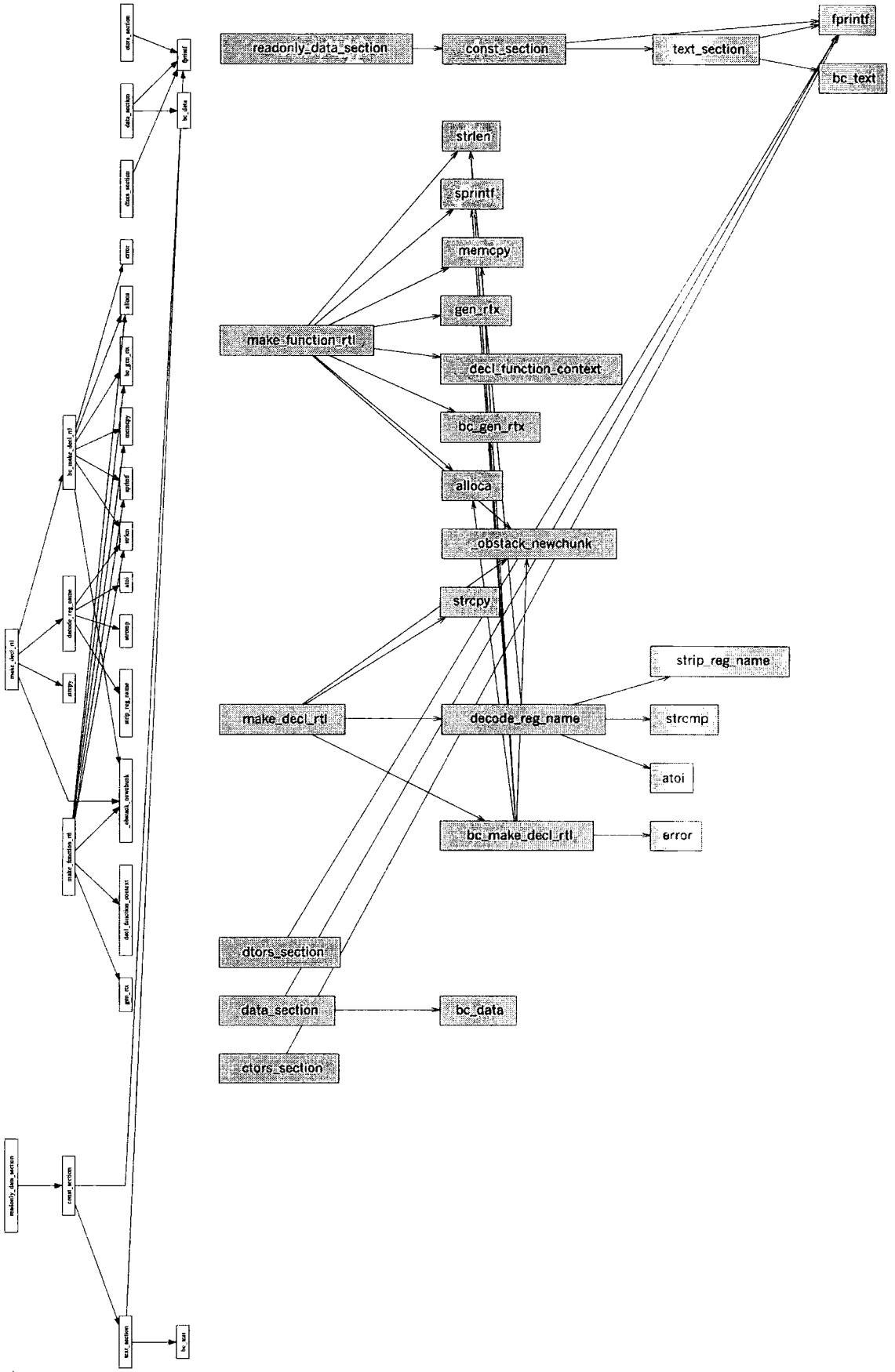
Figure 66 shows how varasm is laid out using conventional tools. Figure 66 shows how it is laid out (a) daVinci and (b) using Graph Tool. They have many problems, such as: -

- the graph structure is not symmetrically recognisable,
- related vertices are not situated close together,
- common structures are not evident, and
- many edges crossings.

It is shown in Figure 67 that the graph is in fact two graphs and this cannot be seen from Figure 66. The illusion of it being one graph is caused in Graph Tool by the error discussed earlier that the layout algorithm cannot deal with vertices that fan into a vertex that has already been placed. For instance the way that vertices labelled 'dtors_section', 'data_section' and 'ctors_section' fan into the vertex labelled 'fprintf'. If the layout algorithm were to be improved then this illusion would not occur. The same vertices cause the illusion in the daVinci laid out graph, however this would have to be solved by improvements in the vertex ordering routine present in the daVinci layout algorithm.

The vertices labelled, 'strlen', 'obstack_newchunk' 'sprintf', 'memcpy', 'bc_gen_rtx' and 'alloc' should all be situated near vertex labelled 'bc_make_delc_rtl'. This would be dealt with using a Split 1 model. Therefore the common structures are not visible in the graphs, and related vertices are not situated closely together.

Figure 67 (a) shows that daVinci produces long and thin diagrams and if printed are difficult to read. daVinci again causes more edge crossings than the other algorithms.



(a) by daVinci (b) by Graph Tool

Figure 66- varasm laid out using conventional graph layout tools.

7.2.3.2 The ANHOF System

The call graph of varasm can be sent through the ANHOF system, using the models provided in Chapter 4. A drawing given in Figure 67 is produced. This has solved many of the problems given above. The common structures are evident and the edge crossings are reduced. The graph is of a reasonable size that it can still be printed. However the vertex problem occurs here, the problem where a vertex cannot be a member of two or more models. Here the vertex labelled 'strlen' cannot be a member of the Split 1 model that it was positioned in and the Fan Out model from the vertex labelled 'decode_regname'. In the case of this diagram it does not hinder overall layout.

Edge crossings could be further reduced if the order of the fan in vertices into vertex labelled 'fprintf' was different. If the position of the vertex 'data_section' was at the top of the vertices or at the bottom then the graph could be laid out without edge crossings.

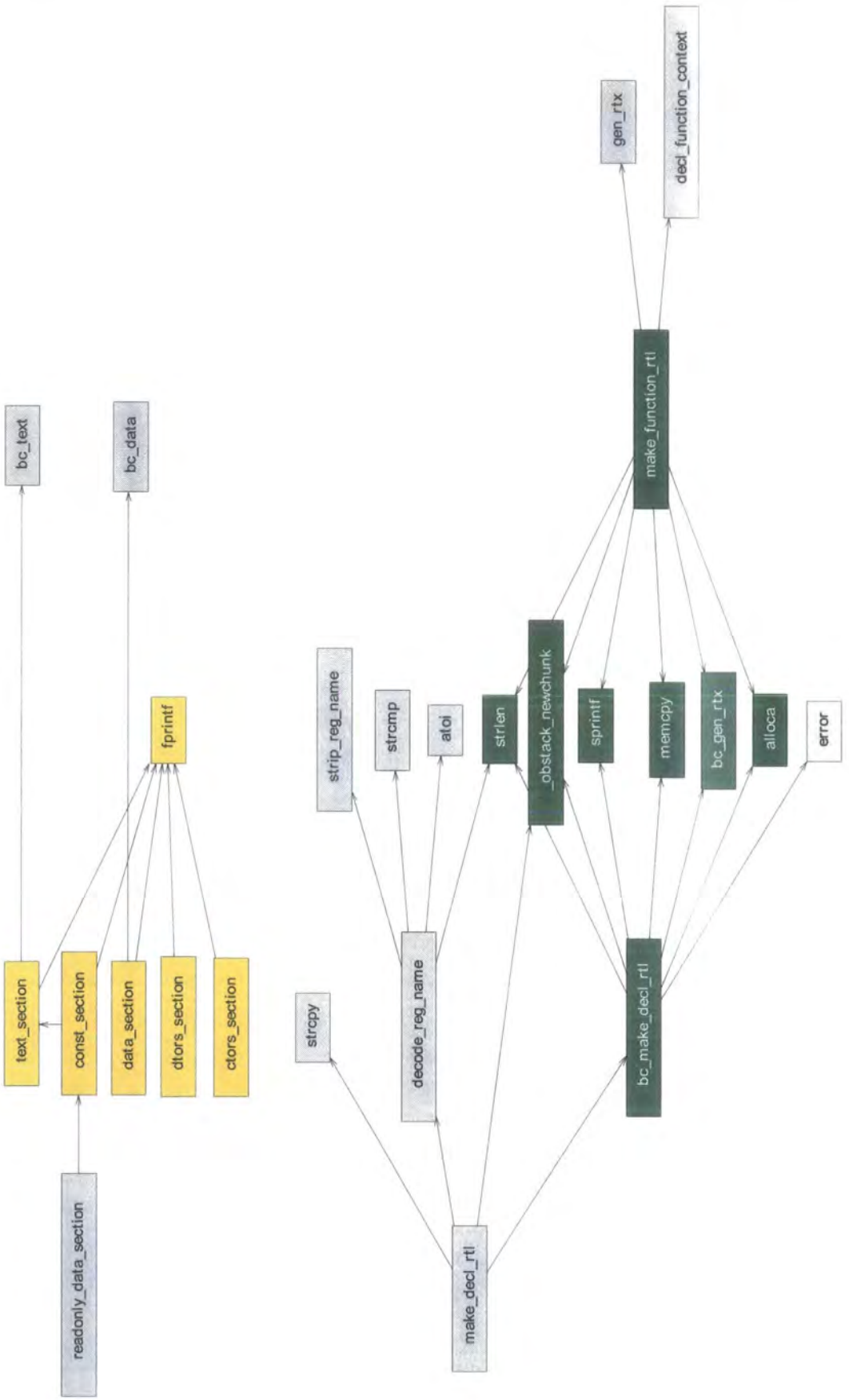


Figure 67- varasm laid out using the ANHOF system

7.2.4 Call Graph of localalloc

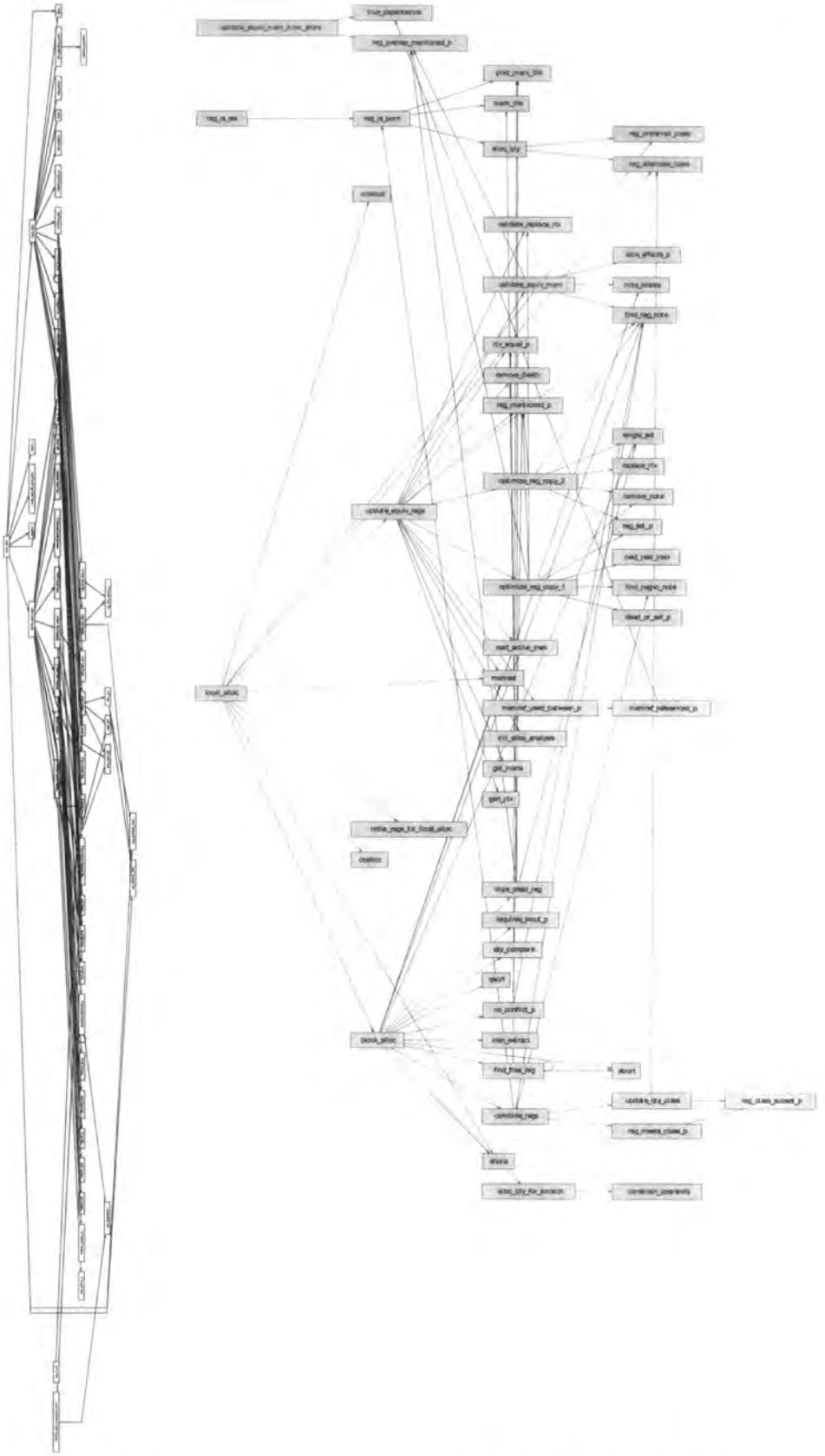
The call graph of localalloc is another program from the GCC source code. It deals with the allocation of the registers so that they can be used within the local areas of the compiling program. The program can be represented as a call graph with 55 vertices and 84 edges.

7.2.4.1 Existing Automatic Graph Layout Algorithms

Large graphs (greater than 100 vertices) are difficult to display on A4 paper because it is difficult to fit all the edges and vertices onto the sheet. Therefore, the call graph of 'localalloc' is amongst the largest graph that can be displayed on an A4 sheet. In Figure 68 the layout gained from using (a) daVinci and (b) Graph Tool is shown. The problems of conventional layouts are obvious in Figure 68 and are: -

- high number of edge crossings,
- related vertices are not placed together, and
- common structures are not close together.

The increasing number of crossings hinders the ability to follow edges in both layouts. Related vertices not being close together causes many of the edge crossings. Most of the common structures are not visible except for Fan Out models. Figure 68(a) shows that daVinci produces long and thin diagrams that are very hard to fit on an A4 sheet of paper. It will be later shown that using Sugiyama's Algorithm, on which daVinci is based, continuously produces long and thin diagrams.



(a) By daVinci (b) By Graph Tool

Figure 68 - localalloc laid out using conventional methods

7.2.4.2 The ANHOF System

Passing the call graph of the localalloc program through the ANHOF system produces a graph laid out in the manner shown in Figure 69. Figure 69 shows that edge crossings, whilst reduced by the ANHOF method / system, are still present in the diagram. They could be reduced further if a better edge routing algorithm could be implemented. The algorithm again produces layouts that are long and thin but less so than Sugiyama. In addition the rule that vertices cannot be part of two or more models is hampering the layout a little. Many vertices are part of two or more models generating an increasing number of vertices that are not involved in models, and therefore increasing the chance of the diagram becoming hard to follow and understand.

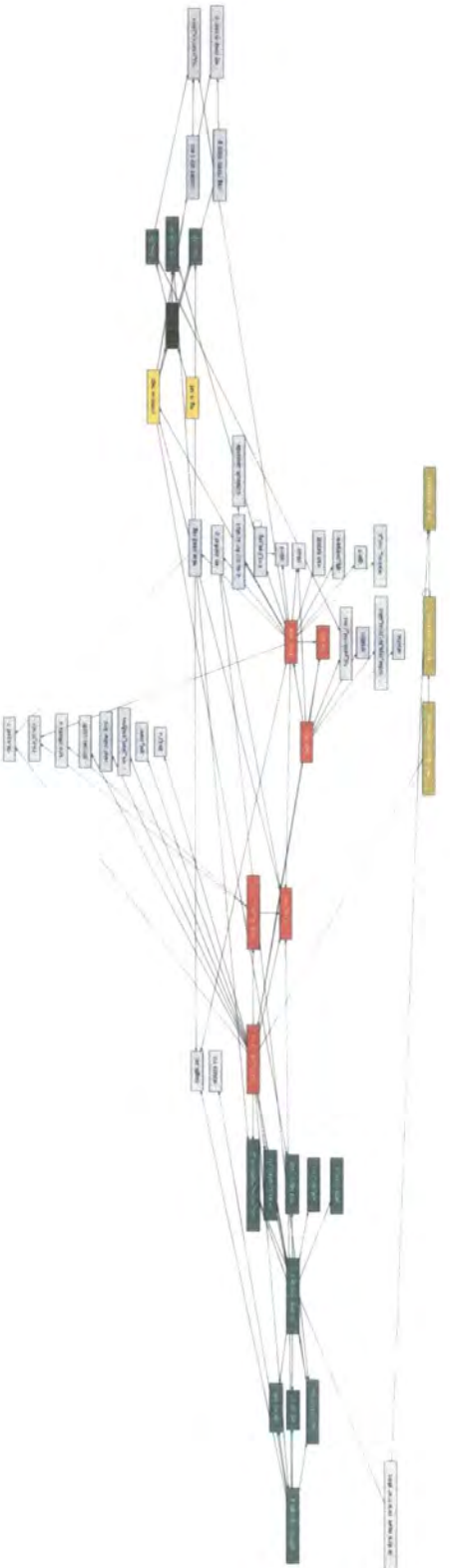


Figure 69 - localalloc laid out using the ANHOF System

7.3 Metric Comparison of the Graphs

In Chapter 2 metrics are given that provide the ability to compare graphs together with methods of calculating them. In Chapter 8 these metrics will be explored and discussed further. In order to compare the graph layouts the following metrics will be calculated: -

- height,
- width,
- ratio of height to width,
- the area occupied,
- the edge length, and
- the number of clusters in the graph.

Program	Algorithm	Crossings	Vertices	Edges	Height	Width	Area	Ratio	Edge length	Clusters
cp-search	ANHOF	0	9	8	161	655	105455	4.07	1687	5
cp-search	daVinci	2	9	8	675	224	151200	3.01	1370	5
cp-search	Graph Tool	0	9	8	185	384	71040	2.08	1349	5
genopinit	ANHOF	1	21	22	500	365	182500	1.37	3882	7
genopinit	daVinci	28	21	22	153	775	118575	5.07	4385	6
genopinit	Graph Tool	11	21	22	677	278	188206	2.44	4467	9
localalloc	ANHOF	73	55	84	733	2648	1940984	3.61	35522	21
localalloc	daVinci	256	55	84	6580	490	3224200	13.43	41220	13
localalloc	Graph Tool	163	55	84	1620	842	1364040	1.92	36179	16
G	ANHOF	0	45	50	462	1005	464310	2.18	6231	24
G	daVinci	36	45	50	514	604	310456	1.18	7094	17
G	Graph Tool	13	45	50	1005	466	468330	2.16	8990	18
varasm	ANHOF	2	26	32	503	849	427047	1.69	6317	12
varasm	daVinci	46	26	32	1564	212	331568	7.38	6548	11
varasm	Graph Tool	11	26	32	759	469	355971	1.62	8656	12

Table 19 - the properties of the example graphs

Table 19 shows the metric properties of the example graphs given above. The layout algorithm of daVinci creates long and thin graphs; these are graphs that have a high ratio between the height and width. However in the case of cp-search the ANHOF system produces the highest ratio and, in the case of the G, daVinci produces the layout that has a ratio closest to one and is therefore the squarest in nature. There are mixed results, which should be subject to further experimentation, the results of which are given in the next chapter. daVinci produces graphs occupying the smallest area, except in cp-search and localalloc, this again suggests further investigation.

Generally the layout algorithm in Graph Tool produces average results, coming first or second in all sectors. There may be a case for extending the layout algorithm with the common structures method. Therefore using the Graph Tool Layout algorithm as the standard layout algorithm in the ANHOF method / system (step 8 in **LayoutRepresentation** (**Algorithm 4**)). This will be investigated in the next chapter. Generally the ANHOF system performs better than Graph Tool, but only just. daVinci was third in the rankings. However the graphs above only provide a small cross section of vertices and edges. The performance of the ANHOF system is not shown on graphs of greater than 250 vertices.

In every case the ANHOF system reduces the number of edge crossings making the graphs easier to follow. Using the ANHOF system, related vertices are positioned together, therefore edge length should be reduced and clustering in the graph should increase. Generally this was case and is the subject of further investigation, the results of which are given in the next chapter.

7.4 Summary

In the above chapter the layouts obtained by sending real and conceived graphs through the ANHOF system are compared with those that are produced using the existing layout tools of Graph tool and daVinci. The layouts produced by the ANHOF system are an improvement over the conventional layouts because: -

- the common structures where easily detected,
- edge crossings were reduced,
- the edge length was reduced,
- the number of vertex clusters increased,
- related vertices were positioned in close proximity to each other, and
- graphs that consisted of two Graphs became obvious.

However the problem of vertices being members of two or more models was shown to be an increasing problem and the sort order of the vertices should be improved. An indication of future problems that could occur if the Graph Tool layout algorithm is used as the 'standard' algorithm is given. The graph tool layout algorithm has great difficulty in dealing with vertices that fan into a vertex that has already been positioned.

It was shown that the ANHOF method/system fixes problems with conventional layouts from the smallest graphs. However conventional layouts could be greatly improved in terms of the metrics of the graph if better edge routing was incorporated. The layout algorithms should not be just a vertex positioning algorithm. It was shown in the graph of localalloc and possibly the graph of varasm that edge routing would improve the layouts produced by the ANHOF system both in terms of the metrics of the layouts and the understanding of the graph. It was shown in all the layout tools the vertex sorting could improve this and further improve the metrics of the graph.

In the next chapter the metrics of the layouts from the ANHOF system are further assessed to find out which order of sorting the matches to the common model graphs and which standard layout algorithm, creates the best layouts in terms of the metrics of the final layout.

8. Performance of the ANHOF System

Chapter 6 suggested how to tune the ANHOF system, which, was then used to compare the layouts obtained using the ANHOF system with those of the daVinci and Graph Tool layout tools in Chapter 7. Chapter 7 also provided areas for future investigation in terms of the metric performance of the ANHOF system. In this chapter the metrics of the resulting graphs from the ANHOF system are further investigated, the types of models that are present in common everyday software are assessed, and the time performance of the ANHOF system is also detailed.

8.1 Models in Software

There are many types of commercial software applications, a few being: -

- **Compiler** – This provides a service to other programs. It converts programs written in text into machine understandable code and is an example of system software.
- **Database** – This restructures information and stores it so that it can be searched for information. It is an example of business software.
- **Embedded Software** – This resides in the read only memory and is used to control products and systems.
- **Text editors** – This allows humans to write and amend text. It is an example of Personal Computer Software (PCS).
- **Graphic editors** – This allows humans to view and edit graphics on the computer screen. It is another example of Personal Computer Software (PCS).

It is often necessary to know what types of graph models are contained within these types of software so that layout algorithms can be tuned. However this analysis is difficult to perform, not only is it difficult to get code in order to analyse, but also it is very easy to make a broad generalisation. The analysis is still of benefit to perform. However it should be noted that the programs are not claimed to be a good representation of the type of program merely an example of them. The programs that are

analysed are given in Table 20. Two databases are analysed as examples of business software, the GCC version 2.58 compiler is analysed as an example of system software, two graph display tool\editors VCG [146] and Graph Tool [16] are analysed as examples of graphic tools in personal computer software and three gas analyser control systems are analysed as examples of embedded software.

Name	Type	Lines of Code	Total Vertices	Total Edges
Cobol Database management	Database	1512	36	394
Flatfile Database	Database	3400	221	489
2006- Gas Analyser Control Software	Embedded	2548	171	315
300 - Gas Analyser Control Software	Embedded	4511	129	376
Network – Gas Analysis Software	Embedded	1724	80	194
GCC version 2.5.8	Compiler	32071	10756	2561
VCG	Graphics	51667	1773	4057
Graph Tool	Graphics	2177	466	760
Elvis	Text	12963	677	791
Emacs	Text	170384	3920	6184

Table 20 - The programs studied

The databases were COBOL programs and the other programs are all C programs. All of the programs except the databases were processed into a set of Prolog facts about them using the processor called CCG system by Kinloch [95]. Included in these facts is the calling information, which can be converted into a call graph. Once the call graph is obtained it can be processed by the Graph Isomorphism System in the ANHOF system obtaining a list of all the models present. The numbers of each model is then calculated. The results are shown in Figure 70. The call graph is obtained from the COBOL programs by recording when paragraphs are performed in that program.

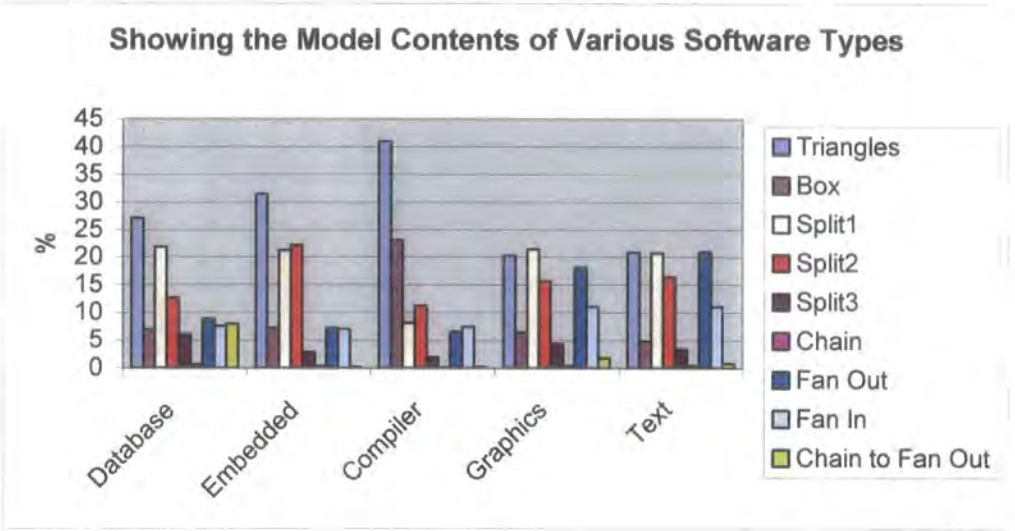


Figure 70 - The average contents of the software tested

Figure 70 indicates that either the Chain models are not common in software or the detection algorithm is poor for them, and that a better definition is necessary. In addition, the compiler has a large percentage of fixed models (Triangle and Box) present in the code, and a lower than average percentage of the other models. Databases have a high proportion of Chain to Fan Out models present. This may be because databases commonly process data that are stored in data types. A Chain to Fan Out model indicates using them, because they are where a procedure calls another procedure that processes information in a data type. Triangles are the most common models found in the code. On closer inspection these are often present in a Fan Out model and was not expected. However, it may be another indication of well-structured code, because a task relies on two further tasks that interact with each other and, in these circumstances code is efficiently used and easily modified.

It was expected that software would have a high proportion of Fan Out models because this indicates that a program is well structured and that a task has been split into many other tasks, each task using a procedure. This was largely the case but was never the most common model found.

If a piece of software has a high proportion of Fan In models this may be a case of the program making high use of standard libraries. Whilst the Fan In models were common

they were not as common as most, therefore suggesting that the use of standard libraries is poor.

In the software a Split 1 model is the most common of the variable models. This is because in programming, two procedures may use the same data but in different ways. It may also be a sign that the code needs restructuring because two procedures may perform the same task. To generalise the results Figure 71 shows the average proportion of models found.

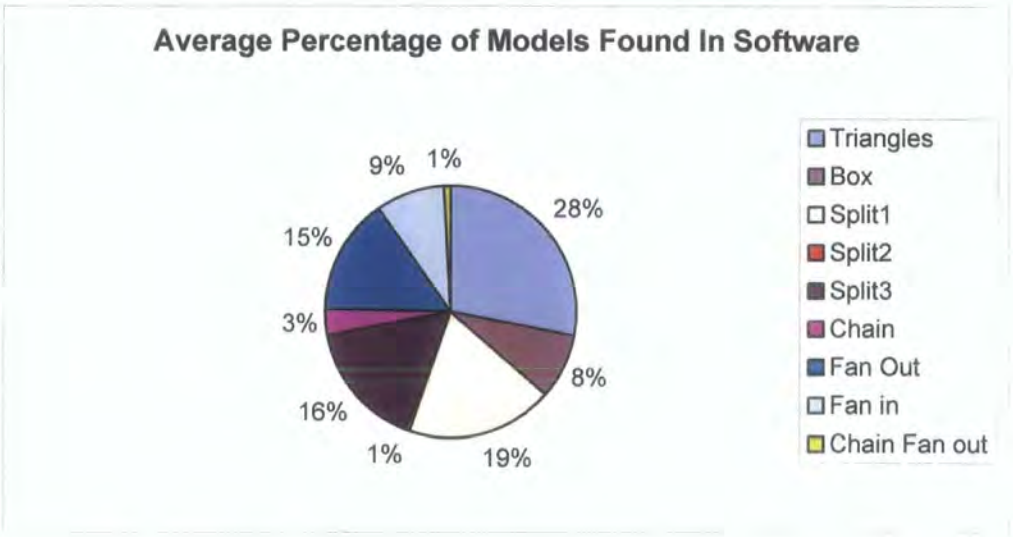


Figure 71 - Shows the average percentage of models in software

8.2 Metric Performance

In Chapter 6 five graphs were evaluated against metrics discussed in Chapter 2. Further discussion is given below, presenting the desired results. There are many papers dealing with the problems of evaluating software engineering methods and tools. Kitchenham and Pfleeger have written a whole series of them, such as [130] and [96]. In terms of graph layout it is a difficult area, the quality of the layout is subjective. However in earlier chapters there are many metrics that can be used to measure the quality of the graph. Call graphs are an aid to program comprehension. To a larger extent this is quantifiable to certain levels. It can then be assessed using Benchmarking. Benchmarking is described by Kitchenham [96] as, "*running a number of standard*

tests/trials using alternative tools/methods (usually tools) and assessing the relative performance of the tools/method against these tests." In terms of the ANHOF system these '*tests/trials*' are a selection of the metrics given in Chapter 2. The following characteristics (metrics) will be measured: -

Area and shape properties: -

- area taken, and
- aspect ratio – ratio between longest and shortest side.

Graph properties: -

- crossings,
- total edge length, and
- the number of clusters of vertices in the graph.

Apart from common structures being clearly visible, generally it is considered that the graph should be the smallest size possible, have an aspect ratio close to that of the output media that it is to be displayed. The edge crossings should be kept to a minimum and edges should be kept as short as possible. Related information should be clustered together and therefore the number of clusters should be high. It is these properties that are strived for in order to create a 'good' layout that is easy to understand. This is what shall be sought in the metric calculations below.

8.2.1 Method of Comparing Call Graphs

The above metrics are applied to 26 graphs from the GCC compiler version 2.58. The properties of these graphs are given in Table 20. Each graph was selected so that either the number of vertices was approximately (within 10 percent of goal) equal to 50,100,150, and 200, or the edges where approximately equal to 50,100,150,200,250 and 300. In general within these graphs each vertex has a Fan Out value of approximately two. Although this may not be true for call graphs in every type of program.

Graph	Vertices	Edges	Graph	Vertices	Edges
aux-output	70	100	genemit	28	45
calls2	89	160	genoutput	32	49
calls2-sub	81	109	genrecog	37	52
c-decl2	151	298	gtest	182	310
combine2-1	112	317	insn-emit-2	151	470
combine3	115	296	jump-2	101	176
cp-class	53	69	loop-2	118	208
cp-cvt	79	149	optabs	89	244
cp-decl2-2	148	207	protoize	94	242
cp-except	110	202	real-2	111	391
c-typeck	147	382	recog	52	70
dbxout	51	84	reload-2	73	156
function-2	151	40	tmp	219	675

Table 21- The properties of the graphs processed

In the following section the results of laying out the above graphs using the methods below will be discussed. They will be compared using the metrics above therefore enacting a benchmarking evaluation method as described by Kitchenham [96]. The results are given in two areas, the area and shape properties and the general graph properties.

Standard Algorithms: -

- graph tool layout algorithm (Standard GT),
- Sugiyama layout algorithm (Standard Sugiyama), and
- Manual (Standard Manual).

ANHOF system methods: -

- Matches as they were detected in order of Split 2, Split 1, Split 3, Chain to Fan Out, Box, Triangle, Fan In, Fan Out, then Chain using Graph Tool as the 'standard' layout algorithm (ANHOF Order using Graph Tool).

- Matches sorted in descending order using Graph Tool as the 'standard' layout algorithm (ANHOF Descending using Graph Tool).
- Matches sorted in ascending order using Graph Tool as the 'standard' layout algorithm (ANHOF Ascending using Graph Tool).
- Alternate matches taken from the order of Split 2, Split 1, Split 3, Chain to Fan Out, Box, Triangle, Fan In, Fan Out, then Chain using Graph Tool as the 'standard' layout algorithm (ANHOF Every other using Graph Tool).
- Matches as they were detected in order of Split 2, Split 1, Split 3, Chain to Fan Out, Box, Triangle, Fan In, Fan Out, then Chain using Sugiyama as the 'standard' layout algorithm (ANHOF Order using Sugiyama).
- Matches sorted in descending order using Sugiyama as the 'standard' layout algorithm (ANHOF Descending using Sugiyama).
- Matches sorted in ascending order using Sugiyama as the 'standard' layout algorithm (ANHOF Ascending using Sugiyama).
- Alternate matches taken from the order of Split 2, Split 1, Split 3, Chain to Fan Out, Box, Triangle, Fan In, Fan Out, then Chain using Sugiyama as the 'standard' layout algorithm (ANHOF Every other using Sugiyama).

The layout of the graphs using the standard algorithms were obtained passing the graph information file (GIN) through the layout algorithms implemented as part of the Graph Layout System. In Chapter 7 graphs were laid out using Sugiyama et al. [159] algorithm, the layout was obtained by using daVinci as the layout tool that applied to algorithm to the graph. In the tests below the layouts are obtained by using the LEDA / AGD library implementations of the various algorithms. The algorithm by Sugiyama et al is a standard layout algorithm contained in the LEDA / AGD library, and the Graph Tool layout algorithm is an implementation of **GraphLayout** (Algorithm 17) using the LEDA / AGD libraries to provide standard functions.

8.2.2 Area and Shape Properties

The ratio of the longest side to the shortest side provides the aspect ratio of the graph. Most paper sizes are rectangular in nature; therefore in order for the graphs to be printed they also need to be rectangular in nature. A square diagram will have a ratio of one, therefore the larger the value the more rectangular the graph is. European paper sizes,

e.g. A4, have a ratio of 1.41 and most displays have a ratio 1.333. Therefore a desirable property of a graph is one of these figures. Figure 72 shows the ratio obtained by the various methods. The applications of Sugiyama's algorithm to a graph tends to cause graphs to occupy a smaller area of paper, but are long and thin, and are therefore difficult to print. This property could be changed using different spacing between vertices. If the value of 1.41 is the desired ratio then it is not achieved, not even by manual layout. However doing it manually did produce the closest. Manual layout was hampered by not being able to evenly space the vertices. Therefore large spaces in the graph were produced that could have been pulled together making the vertex more compact. If manual layout could get over this problem then it would probably come even closer to the 1.41 or 1.33 figure. This was corrected using an automatic algorithm. This however had problems setting the spacing correctly. Using either the Graph Tool Algorithm or the Sugiyama algorithm as the standard layout method in the ANHOF system improves the ratio from using the respective algorithm on the whole graph without the ANHOF system. Results showed that to get the best ratio it is not advisable to sort the matches, but combine them in the natural order (described above and in Chapter 6) and use the Graph Tool algorithm to layout the resulting graph.

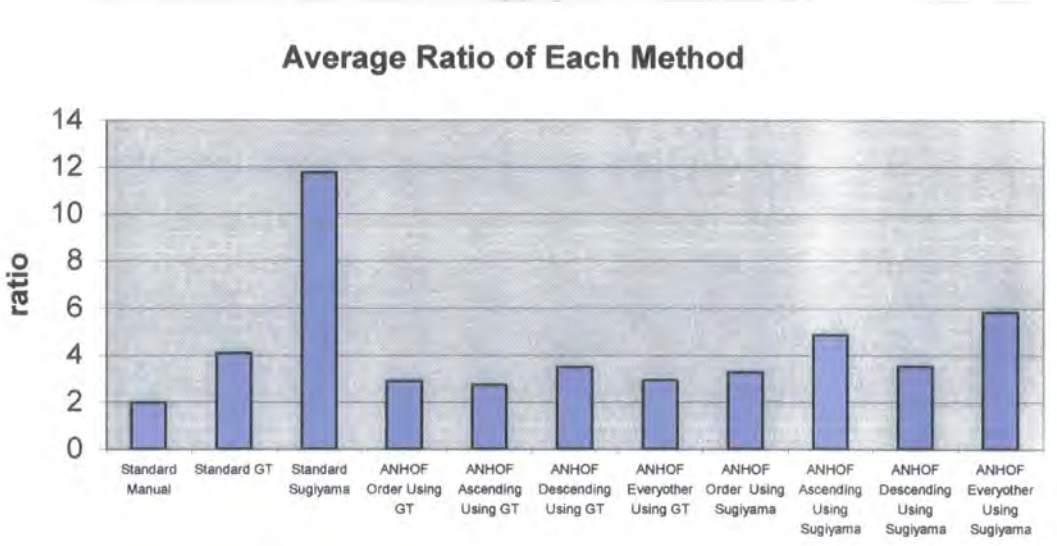


Figure 72 - Shows the ratio between longest and shortest side using the various methods

If the aim is to reduce the area of the graph then the results of Figure 73 suggests the use of Sugiyama's Algorithm. On closer inspection it is long and thin and is therefore

unprintable. The results show that using either the Graph Tool algorithm or the Sugiyama algorithm as the 'standard' algorithm the ANHOF system increases the area taken by graphs. If the matches are sorted into ascending order and laid out using the Graph Tool layout algorithm then its area is polynomial, the others are largely linear. To minimize the area taken using the ANHOF system then the results show that the following should be followed. If the graph has fewer than 65 vertices then the matches should be sorted into ascending order then the resulting graph laid out using Graph Tool. However if the graph is larger than 65 vertices and the area is to be minimized then the matches should be placed in a file in the natural order given above and in Chapter 6 and the whole graph laid out using Sugiyama. However, this is likely to produce a long and thin graph.

A long and thin graph is difficult to follow. Therefore for comprehensible graphs a compromise between the thin graphs produced using Sugiyama and the polynomial performance of the ascending order using Graph Tool should be found. Placing the matches in the natural order described in Chapter 6 and using the Graph Tool layout algorithm as the 'standard' layout algorithm obtains the best performance of the ANHOF system.

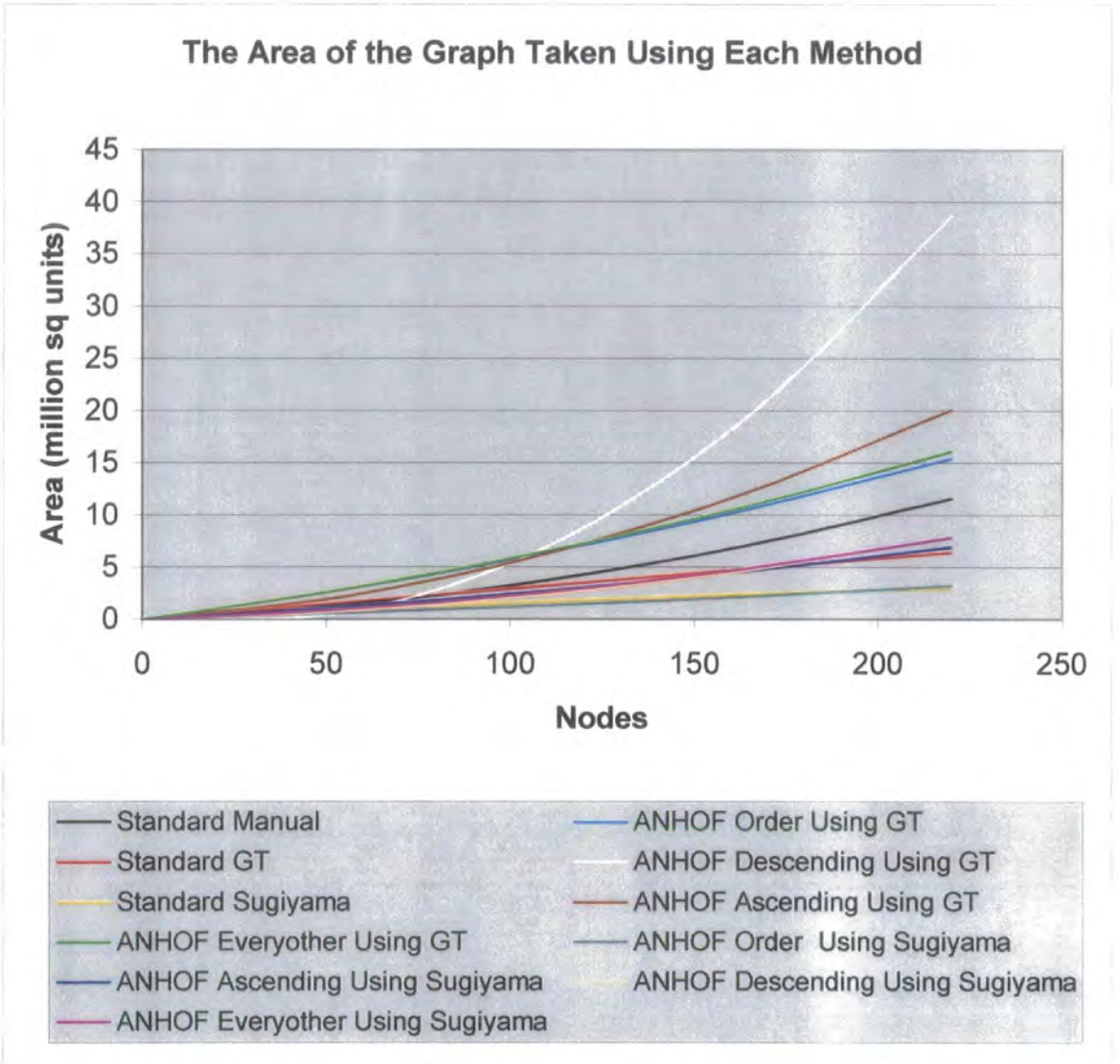


Figure 73 - The area taken by graphs

8.2.3 Graph Properties

It is a desirable property to gather all the common information together in a graph. If in a call graph, a procedure calls other procedures, then the procedures it calls and the calling procedure should be as close together as possible. Chapter 7 suggests that this is not dealt with using a standard layout algorithm. However using the ANHOF system it is possible to cluster, not only procedures that call each other, but common structures in the program that may represent common programming practices. In Chapter 2 a method of calculating the number of clusters in a graph is given based on the distance between

vertices. The relation property (a cluster) used is that vertices are related either though being called by each other or they are a common structure. If they are related through this definition then they should be placed close together. Using the ANHOF method / system should cause there to be a high number of clusters based in the distance on layouts obtained. The results show that this is indeed the case.

Figure 74 shows that the number of clusters is polynomial in nature and that it is a convex curve, meaning that it will peak. This may be because the method invoked to calculate the number of clusters may not be suitable for calculating the number of clusters in large graphs. In Chapter 2 it is shown that calculating the number of clusters is a case of interpreting the Dendrogram at level three. In larger graphs (greater than 150 vertices) the level may have to be level two because the clustering method (nearest neighbour analysis) may cluster graphs earlier in larger graphs than in smaller ones because greater distances are considered. The results show that the number of clusters in a layout drawn using manual methods has indeed already peaked, at around 140 vertices. The results show that the numbers of clusters in layouts drawn using automatic methods are still climbing at this point. The results show that more clusters were evident from layouts obtained using Sugiyama, both as the layout for the whole graph and with the ANHOF system, than those obtained using the Graph Tool layout algorithm, both as the layout for the whole graph and with the ANHOF system. This was because of the method used to cluster them, and was not generally a property of the graphs.

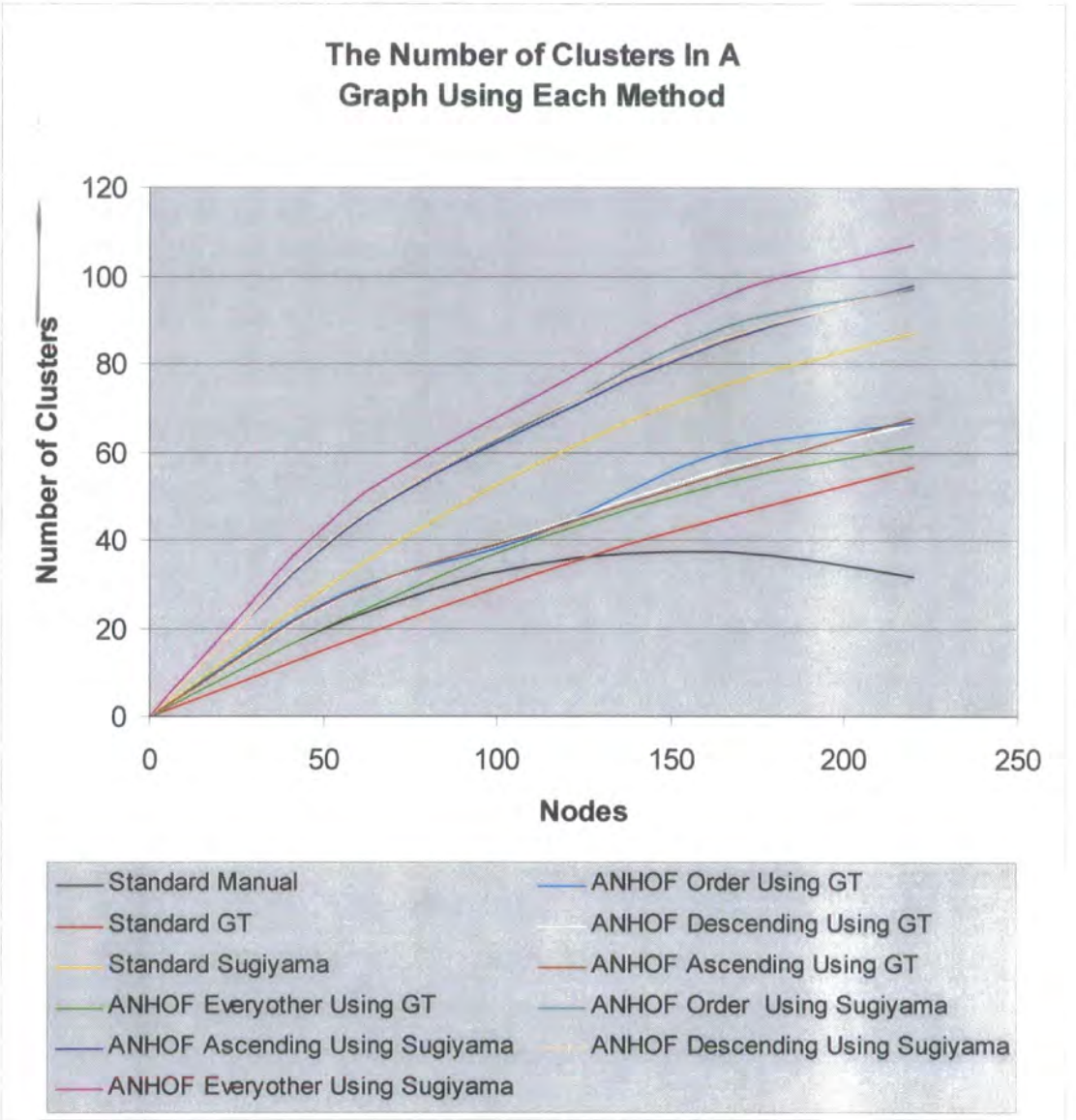


Figure 74 - The relationship between the clusters and vertices

The edge length is related to the area of the graph because if the graph takes more area then the edges will be longer. A large area often means there will be a few very long edges. A long edge can be difficult to follow. The results shown in Figure 75 show that the ANHOF system produces longer edged graphs than normal algorithms. This counteracts the cluster finding, because vertices should be together. It may be because of the spacing used between the vertices. However it is not possible to position all the vertices together, there are always going to be vertices that are positioned by one model layout algorithm but are called by other vertices. These may be positioned at opposite ends of the graph so causing long edges. It may be the result of the rule that a vertex cannot be a member of two or more models. It was shown in Chapter 7 that the ANHOF

system generally reduced the edge length; however investigating this over many more graphs disproved this. Performing the layout manually produces a linear edge length per edge. All methods perform similarly until approximately 310 edges then using the layout algorithm by Sugiyama et al. as the 'standard' layout increases rapidly and causes longer edges than using the Graph Tool Layout algorithm as the 'standard' layout. It has been show earlier that Sugiyama produces long and thin graphs. Therefore the edge length will be increased. Results have indicated that in order to reduce the edge length of a graph when using ANHOF system either sort the matches found into ascending order or place them into the natural order given in Chapter 6 and use the Graph Tool layout algorithm as the 'standard' layout algorithm.

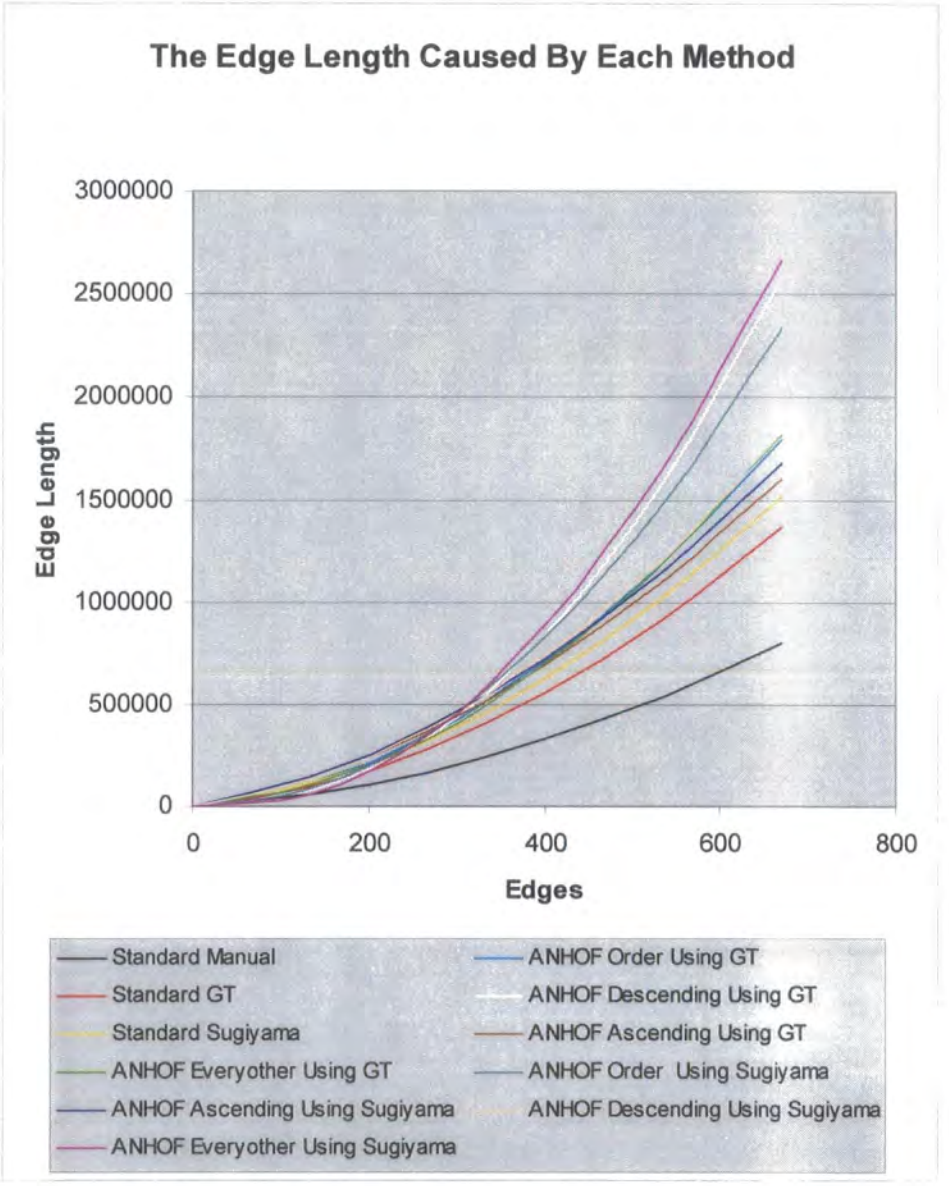


Figure 75 - The relationship between edge length and edges

In Chapter 2 it is suggested that reducing the edge crossings is an important goal for any layout algorithm because it increases comprehension. In most automated graph layout algorithms edge crossings are inevitable. The ANHOF system is no exception. Figure 76 shows the results of counting the edge crossings in each graph. The results have indicated that using Sugiyama's Algorithm seems to produce more crossings than the Graph Tool algorithm. All methods produce similar results until around 200 edges, from this point onwards most increase rapidly at different rates. The ANHOF system produces fewer crossings than the standard algorithms. The results show that using the natural order described in Chapter 6 and using the Graph Tool algorithm produces the fewest crossings in every case. It is also more linear than the others. The results also shows that sorting the matches into ascending order and using Sugiyama as the 'standard' layout algorithm, and taking every other match from a list of matches in the natural order (discussed in Chapter 6) and using Graph Tool the 'standard' layout algorithm produces less crossings than using the conventional layout algorithms of Sugiyama and Graph Tool.

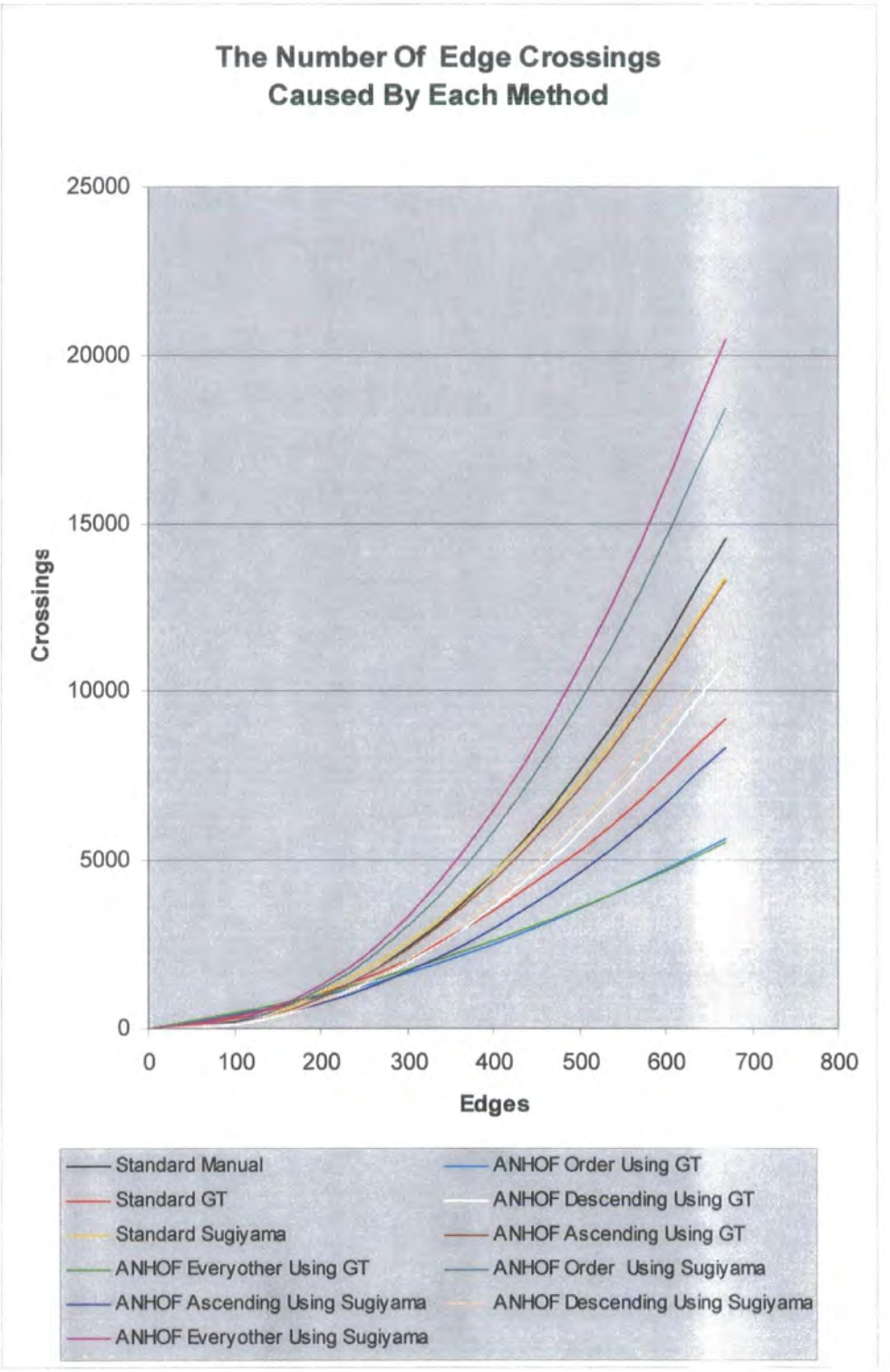


Figure 76 - The relationship between the number of crossings and the number of edges

From all the above comparisons of graph metrics, it is evident that one method of graph layout performs consistently better. The results show that the ANHOF system should be used in the following manner. Search for the common model graphs creating nine

separate lists of matches. Then combine the list of matches in the order Split 2, Split 1, Split 3, Chain to Fan Out, Box, Triangle, Fan In, Fan Out, then Chain into one large list. Pass this list of matches through the Match Analyser, and through the Graph Layout System using Graph Tool's layout algorithm as the 'standard' layout algorithm. This will maximise the performance of the system and produce a graph that may aid program understanding.

8.3 Time Performance

It is desirable for a program to execute in linear time. Figure 77 shows that the ANHOF system largely does execute linearly, one vertex taking approximately three quarters of a second to place in total. This figure is not desirable in terms of an interactive graph layout system. The ANHOF system is a prototype system that was designed to show the concept and Chapter 7 shows that the concept works. A great deal of time is spent in the ANHOF system outputting status information to the screen. If this was removed then time could be saved. The sorting algorithms used in the Graph Layout System are largely inefficient and more efficient algorithms could be written, again offering an optimisation of the system.

Any deviations from the linear execution time are caused by it taking longer to lay out a graph that has a high percentage of common structures than those with not. Two models take more time to layout than one whole graph. This is due to the fact that a model is laid out first and the vertices contained in the model are all combined into one vertex in a new graph. This new graph is then laid out using a 'standard' layout algorithm before the vertex representing the common model graph is expanded again. This of course is going to take longer than just sending the whole graph through the 'standard' layout algorithm in one go. Many of the graphs used in this evaluation consist of numerous Fan Out models. However it is better if the 'standard' layout algorithm evaluates these, because laying them out as separate models causes additional overheads in time. It would be an interesting experiment to investigate if removing the Fan Out model from the library improved the time taken and the quality of the graph.

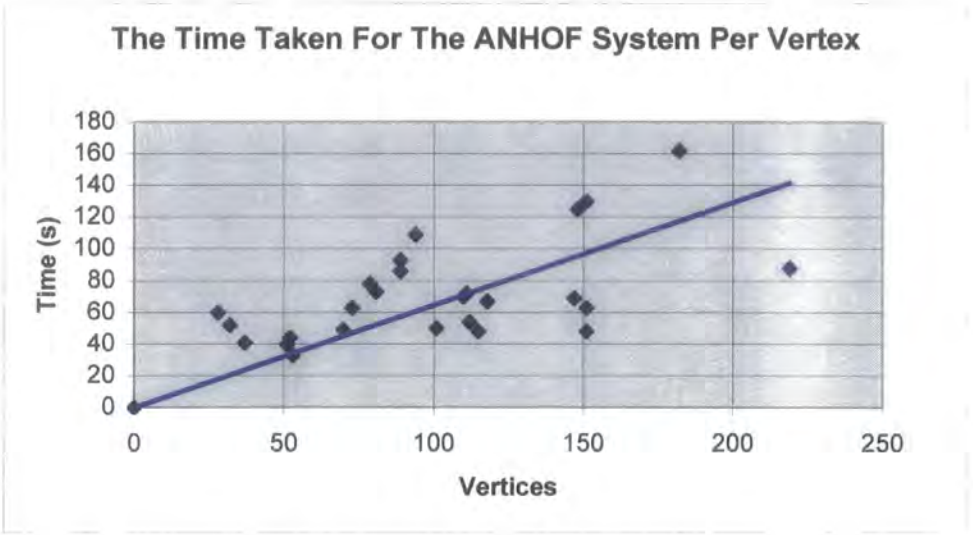


Figure 77 - The running of time of the ANHOF system

The results shown in Figure 78 show the percentage time taken by each of the three processing steps of the ANHOF system. The two most processor intensive tasks (the Graph Isomorphism System and the Graph Layout System) take the major share of the time taken. It is interesting to observe that the Graph Isomorphism System takes up 50 percent of the time. It is thought that the Graph Layout System should take up most of the time because this is possibly the most processor intensive task. It is also interesting to note that the fixed isomorphism detection system ran quite efficiently taking only 12 percent of the time in the Graph Isomorphism System or six percent of the total time, whereas the Variable Model Detection System takes 88 percent of the time taken by the Graph Isomorphism System or 44 percent of the total time. However, 21 percent of the time taken to detect the variable models is taken in creating the Prolog representation of the input graph and calculating the fan in and fan out information. This is good area for optimisation; two tools that could be easily combined currently perform these. One reason why the Fixed Model Detection System runs so efficiently is that one program searches for all the models, therefore the graph is stored once and the models are read only once. This is more efficient than the Variable Model Detection system where each model is detected using a separate Prolog rule executed sequentially. Each rule having to be executed and graph and fan information loaded, this is a very inefficient method of performing this step. Prolog, whilst efficient at searching the information, has a large

initiation time that the Fixed Model Detection System does not suffer from. These are the areas that could be optimised.

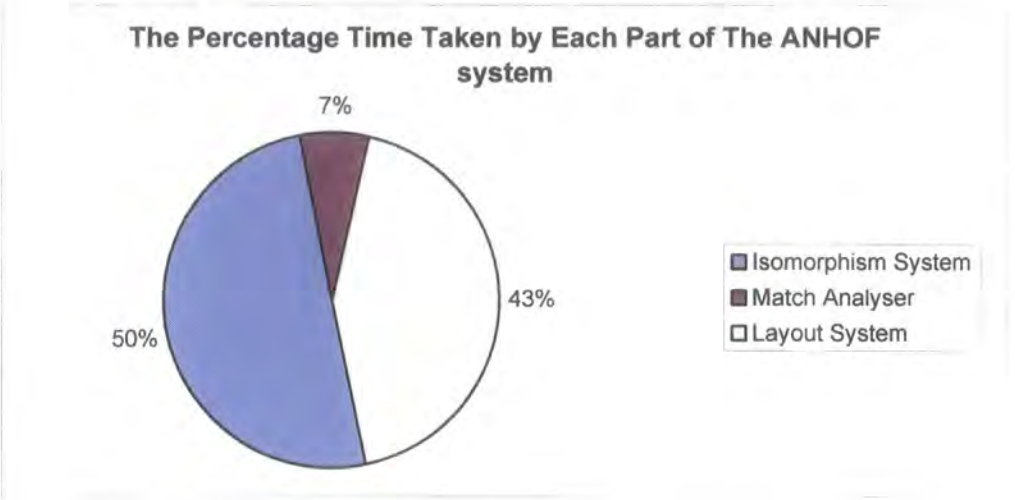


Figure 78 - Where the time is spent in the ANHOF system

8.4 Summary

This chapter provides analysis of the types of models that are found in everyday computer programs. This is generalised further by providing the common model graph contents of computer software. The ANHOF system was evaluated in terms of the metrics of the graph it produces. Finally the time performance of the ANHOF system is calculated. The percentage of time spent by each part of the system is given. In the next chapter conclusions are drawn from this research and further work is detailed.

9. Conclusions and Future Work

9.1 Introduction

This chapter presents a summary of this thesis and evaluates the success of the research work against the criteria defined in Chapter 1. Comparisons of the ANHOF method with a similar method and possibilities for further work in the future are also discussed.

9.2 Background

Graphs are used in many every day tasks from modelling interactions between particles in chemistry to designing circuits in engineering. Laying out these large graphs is a difficult, labour intensive task. It is a task that is ideal for automation. The algorithms to perform this are known as automatic graph layout algorithms and are subject to a great deal of research, and form the main part of the theory behind this thesis.

This thesis presents research into the application of graph layout techniques to the automatic layout of software engineering graphs. These graphs quickly become unreadable because edge crossings are high and vertices often overlap. Application of many 'standard' layout algorithms may reduce the overlapping vertices but the edge crossings are still high. The work carried out for this thesis tries to improve the layout of software engineering graphs.

Software engineering concerns the process of producing computer programs for use in everyday tasks. When software is written it is often produced so it that can be used for many years. Within these years it is inevitable that the software will have to be changed to meet new requirements. This is known as software maintenance. This is why, when modelling the software engineering process, maintenance is a large proportion of the process. Often software is maintained that is many years old, the software was often programmed before the advent of software engineering methods, the original

programmers have left and the documentation is non-existent or has not been kept up to date. Therefore the programmers that have been given the task of implementing the change to the software face a daunting task of understanding the code. This is where the field of program comprehension is used. This provides methods of modelling a programmer's mind, allowing tools to be developed that aid the programmer in understanding the code. One such method is to use a visualization of the program. Visualization may take many forms and represent many aspects of the software. One such form may be a two dimensional graph that represents the flow of data through a program (data flow graph) or a graph that represents the calling relationship between procedures or functions (call graphs).

There are two approaches to automatic graph layout algorithms. One is the algorithmic approach consisting of designing special purpose layout algorithms, each algorithm devoted to solve the layout problem to specific sets of requirements and specific graph structures. Another is the declarative approach consisting of devising languages for describing requirements, and of using logic programming to construct diagrams that fit the given requirements. There is a growing trend that implements a combination of the two.

Since the early 1980s there has been an impressive growth in the number of automatic graph layout algorithms. Automatic Graph Layout Algorithms tend to work well for small graphs (less than 50 vertices) and do not scale up well to larger graphs (greater than 150 vertices). Also the algorithms usually apply to specific classes of graphs. The graphs in the domain of software engineering tend to consist of many classes of graphs. Consequently the automatic graph layout algorithms provide poor results. It is a view that research in automatic graph layout cannot proceed further until this issue is addressed, and layout algorithms are developed that work on specific types of graphs. This research improves the layout of a specific type of software engineering graph known as the call graph.

When comprehending programs maintainers tend to chunk section of code together. They also look for beacons of code that indicate the presence of certain processes. These techniques can be applied to call graphs. These beacons and chunks tend to correspond to the presence of certain common structures in call graphs. When

understanding call graphs maintainers look for these common structures. When using standard layout algorithms these common structures do not become prominent and therefore the graphs do not aid comprehension and the point of the graphs has been lost. The common structures can also be used to improve the layout of graphs if the common structures have an associated good layout that aids understanding and reduces the number of crossings in the graph. The common structure in the form of a graph and its associated layout algorithm are collectively known as common model graphs. The original call graph is broken up into these common model graphs and other graphs known as subgraphs. Each subgraph is then laid out using its associated layout algorithm and the graph is then rebuilt of well laid out graphs. The layout that is left is greatly improved. Also because the graph has been broken up into smaller subgraphs the standard layout algorithms yield better results.

The following section provides a discussion of the work presented within this thesis, identifying what has been accomplished. This is followed by an evaluation of this research work against the criteria for success given in Chapter 1. Finally, possibilities for further work and future directions of this research are identified.

9.3 Results

In this thesis a method of graph layout, know as the ANHOF method is presented. This is a four-part method for automatically laying out call graphs; these are graphs used in software engineering. Within many call graphs there exist common structures. These structures are laid out in the same manner every time, and therefore become recognisable and aid comprehension. These common structures are given a standard layout and are know as common model graphs. The ANHOF method allows these common model graphs to be described and searched for in the call graph using subgraph isomorphism.

Each part of the ANHOF method is discussed and detailed. The four-part process consists of three processing parts and one display part. The common model graphs are detected using the Graph Isomorphism System. This produces a list of matches that are then filtered by the Match Analyser. The Match Analyser applies the rule to filter the

models, that a vertex cannot be a member of two or more models. The Match Analyser produces a list of valid matches that are incorporated in a graph representation and used to layout the graph using the Graph Layout System. The final layout is displayed on a Graph Display System. In Chapter 4 specific algorithms are given that can be used to implement each part.

A proof of concept implementation of the ANHOF method has been developed. The implementation is known as the ANHOF system. It implements each part of the method as a separate program. Two programs detect the two types of common model graphs in the Graph Isomorphism System, one for the fixed common model graphs, known as the Fixed Model Detection System and one for the variable common model graphs, known as the Variable Model Detection System. Details are given of all the languages used to implement the ANHOF system, giving a description of the languages used to describe the models, the layout algorithms and the aesthetics

This thesis evaluates the ANHOF system; the settings for tuning the ANHOF system are given. These are settings that are necessary to get the maximum performance out of the ANHOF system. The lengths of chains and the fan out and fan in levels necessary to detect the maximum number of models by the Variable Model Detection System are detailed. The results indicate that Ullman's algorithm is the most suitable for detecting the maximum number of models by the Fixed Model Detection System. The results of experimentation into the methods of getting the maximum number of matches through the Match Analyser are detailed and a natural order for the matches to be combined is discussed. The natural order is shown to improve the metrics of the graph. Methods of sorting vertices in a graph so that the edge crossings are minimized are also discussed. The vertices need to be sorted because many layout algorithms traverse hierarchies so that the next vertex is the next unvisited vertex. The next unvisited vertex is often the first on a list. This list can be sorted in many ways; it is these orders that are discussed. Finally the spacing of vertices on the horizontal and vertical plane is given so that the ratio between the longest and shortest side of the bounding box of the graph is minimized.

This thesis compares the layouts obtained from the ANHOF system with those obtained from tools implementing standard layout algorithms. The tools are daVinci, which

implements Sugiyama's layout algorithm and Graph Tool that implements **GraphLayout** (Algorithm 17). Existing layout algorithms have many problems that are all corrected by the ANHOF system. The following problems were detected in this thesis: -

- the related vertices are not situated with each other,
- the common structures are not apparent,
- high number edge crossings,
- the hierarchy is difficult to follow.
- some edges are lost, and
- often the graph structure is not symmetrically recognisable.

The results of experiments into the models that are found in software are discussed. To generalise the results indicate that software consists of 28 percent Triangle models, eight percent Box models (36 percent Fixed models), 19 percent Split 1, one percent Split 2, 16 percent Split 3, three percent Chain, 15 percent Fan Out, nine percent Fan In and one percent Chain to Fan Out (64 percent Variable Models). The layouts obtained by the ANHOF system are evaluated in terms of their metrics. Finding that the ANHOF system is successful because: -

- it reduces the ratio between the longest and shortest side of the bounding box,
- increases the number of clusters in the graph, and
- decreases the number of edge crossings.

These metrics are improvements because if the ratio is reduced then the graph is easier to print and display, because it becomes closer to the 1.41 ratio of European paper and the 1.33 ratio of the display screen. If the number of clusters in the graph is increased it corresponds to the related vertices, which are situated together, and the common structures, which are increasingly present. Edges are easier to follow if they do not cross, therefore reducing them aids in the comprehension of the graph.

It is also shown that the ANHOF system is not successful in: -

- reducing the area taken by the graph, and
- reducing the edge length

This thesis also evaluates the time taken to perform the graph layout process using the ANHOF system. It found that it takes approximately three quarters of a second per vertex to place each vertex. This is not ideal if the Graph Layout System is to be interactive but adequate given the circumstances. It shows that 50 percent of the time is spent in the Graph Isomorphism System; the efficiency of which can be improved and a method of doing this is given.

This thesis shows that maximum performance of the ANHOF system, in terms of producing the best metrics system, is achieved by forming nine separate lists of matches. The lists should be in the order they are detected. Combine the lists in the order Split 2, Split 1, Split 3, Chain to Fan Out, Box, Triangle, Fan In, Fan Out, then Chain to form one list. The list of matches is then passed to the models through the Match Analyser, yielding a representation of the graph that is then passed through the Graph Layout System. The Graph Layout System should use the layout algorithms associated with the models and the Graph Tool layout algorithm given in **GraphLayout** (Algorithm 17) using the spacing between the vertices of 100 on the horizontal plane and 50 on the vertical plane.

The above order in which to combine the list of matches to the common model graphs shows that in terms of the metrics it is better to have many vertices involved in few models. Therefore it is better to have larger models present in the final layout that use more vertices. This may not be a result that helps comprehension, as it is foreseen that more model graphs prominent in the layout is better than few.

9.4 Evaluation Against The Criteria For Success

The criteria for success, given in Chapter 1, are again given below in addition to a brief evaluation of how this research has addressed each criterion.

- **Identify the common structures in call graphs.**

In Chapter 4 the common structures that are present in call graphs are discussed. Earlier works by Munro et al. [117] found five common structures present. The five models found are all known as primitive models. Nine structures that are present in call graphs are given in Chapter 4, these consist of the original five and four new ones. The four new ones are variations of the primes, added to ease the searching for models. The chapter shows that there is two types of models present in call graphs, fixed and variable. Fixed common model graphs consist of a strict structure; they have a fixed number of vertices, edges and edge direction. Variable common model graphs consist of a variable number of vertices and edges, but have a common edge direction and structure. There are seven variable models (named Fan In, Fan Out, Chain, Chain to Fan Out, Split 1, Split 2, and Split 3) and two fixed models (named Triangle and Box). The chapter also shows that various common model graphs are made up of primitive common model graphs. These are common model graphs that cannot be simplified. There are five primitive common model graphs, two fixed common model graphs (named Triangle and Box) and three variable common model graphs (named Fan In, Fan Out, Chain).

- **Produce well laid out call graphs that are to a high quality described in metric criteria.**

In Chapters 7 and 8 it is shown that the ANHOF system produces graphs that are an improvement on the layout obtained from using existing automatic graph layout algorithms. They are an improvement because the common structures are apparent and the related vertices are situated together. The number of clusters metric shows this, which is greater than the number in 'standard' layouts. In the ANHOF system the metric calculating the number of edge crossings shows that the number is decreased, also that the rate of increase is more linear than the other algorithms.

- **Be able to improve the layout of large call graphs with greater than 150 vertices.**

In Chapter 8 it was shown that the above improvements were made in graphs that have between 200 and 250 vertices. There is no reason why this should not continue to be the

case for graphs having greater than 300 vertices. All the detection methods and layout tools can easily cope with larger graphs.

- **Have the ability to describe the graph in a simple language.**

In Chapter 5 four languages are discussed. Each language is for a separate task; one is for the simple aesthetics to be described, another is to describe the layout algorithms and the third is to describe the common model graphs in terms of the adjacency matrix or the fan in and fan out information. A simple language that allows the whole graph to be represented and also the common structures to be given is the last to be described. The name of layout algorithms to be used to layout the whole graph and common structures can also be stated. The language is reasonably easy to understand and is in simple ASCII format so that it can be read by both machine and human.

- **Be able to detect various common structures that have been found to be present in many call graphs.**

In Chapter 4, algorithms are given that detail how the Graph Isomorphism System should detect the common structures. In Chapter 5 it is shown that the two types of common model graphs are detected by two programs and in two different ways. Variable models are detected by the Variable Model Detection System that is a Prolog based system and applies logic rules to the fact bases about the graph to search for the common model graphs. The Fixed Model Detection System detects fixed models; this is a program that implements various standard subgraph isomorphism algorithms, the best being Ullman [164]. This is an adaptation of Messmer's [114] Graph Matching Toolkit.

- **To develop a prototype tool to show poof of concept.**

In Chapter 5 an implementation of the ANHOF method is given that and is know as ANHOF system. It consists of a four part system. The system is largely implemented in C++, although each part is implemented in different ways. The common model graphs are detected by the Graph Isomorphism System, which is in fact two separate programs, one to detect the fixed common model graphs written in C++ (known as the Fixed Model Detection System) and another to detect the variable common model graphs

written in Prolog (known as the Variable Model Detection System). This produces a list of matches to the common model graphs that is filtered by the Match Analyser, which produces a representation of the graph. This representation is then laid out using the Graph Layout System that can use standard automatic graph layouts or those for the models to layout the graph. This then produces a file that can be displayed on Graph Tool.

It can be seen above that the research has achieved all of its criteria. Overall the ANHOF method / system has been successful. It demonstrates that subgraph isomorphism is a successful method of laying out a graph. Isomorphism is used to find common model graphs in the whole graph that can be used to improve the layout of the whole graph. In the proceeding section the ANHOF method is compared against a similar theoretical system that uses a similar method to layout a graph.

9.5 Comparison Of The ANHOF Method / System With Other Systems

A similar method of laying out graphs with a pre-specified layout was described by Kosak, Marks and Shieber [100]. Here the input graph and the pre-specified layout are given in the form of a language grammar, these is then processed using Prolog. The grammar was similar to that of a layout grammar discussed in Chapter 3. It allowed only fixed common model graphs to be described and not variable common model graphs. Variable common model graphs form the majority of the common model graphs discovered in call graphs. Not all graphs can be represented using the grammar-based system, whereas in the ANHOF method all directed graphs could be laid out. However Kosak et al. [100] suggested with some interesting problems of using such a method to layout graphs. The ANHOF method addresses and answers the problem identified below.

1. **No guarantee of success** - A graph may not possess the specified common model graphs

2. **Interacting common model graphs** – There are potential problems with vertices being members of two common model graphs.
3. **Occasional unacceptable performance** – Prolog backtracking is occasionally inefficient when a solution is hard to find.
4. **Introduction of unwanted common model graphs** – There is no guarantee that correct common model graphs are discovered and the valid common model graphs are the best to use.

The common model graphs in Kosak et al. were fixed and therefore there was a chance of them not being present in a Graph. The ANHOF method combines both fixed common model graphs that may not be present in the graph with variable common model graphs. These variable common model graphs are described in such a way that if the associated parameterised values are set correctly some if not all the variable common model graphs should be present. For instance a Fan Out common model graph is basically a hierarchy and a call graph is naturally a hierarchy. If no models are found then the graph is laid out using standard layout algorithms, whilst there is no improvement over standard layout algorithms the graph will at least be laid out with a reasonably successful layout algorithm. This was not the case in Kosak et al.

The interacting common model graphs are solved by the ANHOF method by simply not allowing them. The main layout algorithm catches any vertices that are not part of the common model graphs.

The performance issue is hard to evaluate. Certainly most time is spent by the ANHOF system in the Graph Isomorphism System. This may or may not be because of Prolog. However not all of the Graph Isomorphism System is implemented in Prolog. Ullman's algorithm can take a long time to search for the models. The actual detection of each model does not seem to take very long, taking an unnoticeable amount of time. It is the execution of the Prolog language that does. It takes a long time to load up and begin processing. It was found that the backtracking was efficient.

It is hard to say whether unwanted common model graphs were introduced in the ANHOF system. There is certainly no notion of an unwanted model in the ANHOF method any model that finds its way to the final layout will aid comprehension because

it is what the maintainers are looking for. The Match Analyser reduces the matches found by 97 percent, reducing a graph with 600 matches to the common model graphs to single figures. There is no guarantee that these models are the best, however it is shown in Chapter 6, 8, and 9 that the ANHOF method is successful and therefore the models seem to be correct.

9.6 Future Work

It is shown above that the ANHOF method of call graph layout has been successful. It has met all of its criteria of success and has drawn together many fields of graph theory and layout. However there are many areas in which this success can be increased. Shown below are improvements to the three processing parts of the ANHOF system, the system that implements the ANHOF method. Also it discusses improvements to the whole theory and method that is behind the ANHOF method.

9.6.1 Improvements to the Graph Isomorphisms System

There are two parts to the Graph Isomorphism System, one for the fixed common model graphs and one for the variable common model graphs. The Fixed Model Detection System works generally well. Little can be done to improve it. The system works efficiently and well. However because the implementation is an adaptation of Messmer's [114] Graph Matching Toolkit it is needlessly complicated. It may be better if it was implemented in another system, say Prolog, whose backtracking is very suitable to the purpose.

The Variable Model Detection System again worked successfully, it may have been hindered, in terms of performance, by the fact that each model was implemented by one rule. It therefore may be better if that was implemented as one. As graphs get larger and more complicated the Prolog rules that search for the matches to the variable models may have to make increasing use of the graph fact base, the fact base that stores the raw vertices and edge information, or the information stored in the fan information fact base may have to be improved. This is because the fan information, stored in the fan information fact base may be proved unreliable. For instance when a graph edge is bi-

directed, i.e. a graph goes to a vertex and back again, it appears twice in the fan in and fan out information once in the fan in and once in the fan out. This may mean that various models are not detected properly. In Figure 79 the vertices are labelled with their name and a tuple (fan out, fan in). This figure represents a valid split 1 model and the layout algorithm should layout the model. But the question is it a valid split 1 model and would the maintainer still recognizes it as a split 1 model? If the answer to this question is no then the edge information in the graph fact base will have to be used. The same is true for self-referencing vertices.

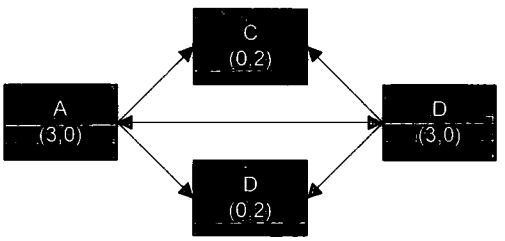


Figure 79 - An example Split 1 model

9.6.2 Improvements to the Match Analyser

The actual Match Analyser worked very well. It was shown in Chapter 8 that to improve the metrics of the graph, then more vertices should be involved in few models. However this may not aid comprehension of the graph where more instantly recognisable common structures should be present in the graph layout. This means that many vertices should be involved in many models. At the moment the Match Analyser reduces the number of matches by 97 percent, reducing a list of 600 matches to single figures of valid matches in most cases. This makes it hard to investigate whether the models being present will aid comprehension of the program. A method of maximising the number of valid matches that are produced by the Match Analyser was investigated in Chapter 6. But this increases the number of common model graphs by single figures, not tens or hundreds. The problem is the rule that the Match Analyser applies. If a method could be found that allowed a vertex to be a member of two models, either by laying every model match that was found or by some filtering technique then more common structures (common model graphs) will become prominent in the final layout. There are

many methods of increasing the number of models used to layout the graph. One is to restrict the number of vertices that can be involved in a model, either by not allowing them or removing the larger models (Split 1, Split 2) from the models that are the focus of the search. Therefore smaller models are used that involve fewer vertices. Another is to apply the rule that if a vertex is a member of two models then delete it from one. All of the vertices in the two models are laid out using the smaller 'good' layout algorithms. There may be many more methods of increasing the number of valid model matches, involving many vertices in common model graphs.

9.6.3 Improvements to the Graph Layout System

The Graph Layout System again works well. There are various efficiency gains that could be performed to make it run faster. These include using more efficient algorithms for sorting vertices and edges and making better use of the data structures in LEDA. However these are superficial. In terms of the layout it produces there are two improvements that could be tried. It may be a view that the number of edge crossings cannot be reduced further unless edge routing techniques are introduced. At the moment the edges are straight lines. Some say however that lines with kinks are not as easy to read. Maybe bezier curves could be used that bypass the obstruction or some variation. If the Graph Tool Layout Algorithm is to be used then the improvement suggested in Chapter 7 should be implemented. This is when traversing the hierarchy and all the vertices that flow from the current vertex have been visited then search for unvisited vertices that fan into the vertex and is illustrated in Figure 63. This may improve the layout algorithm greatly. The ANHOF system implements very few aesthetics into the system and more could be implemented, e.g. symmetry.

9.6.4 Overall Future Work

Generally the suggested work above is minor improvements to the current ANHOF system. There are many areas that could further enhance this research. A few are: -

- to prove that increasing the number of common model graphs actually increases the understanding of the graph,
- to prove that the layouts used to layout the common model graphs actually aid comprehension,
- to try other layouts to improve the metrics still further,
- to try the technique on very large graphs (greater than 1000 vertices),
- to increase the number of common model graphs that are search for,
- to try the method on other types of software engineering graphs,
- to improve all the languages, and
- to incorporate more aesthetics into the layout.

Proving that increasing the number of common model graphs will actually aid comprehension is a very difficult and important task. If it is found to be false and it is necessary to improve the metrics of the graphs then this research will contribute to improving the layout of call graphs. If it was found to be true then there is considerable work to be done on improving the Match Analyser.

This method has been shown to be successful and therefore should be applied to other software engineering graphs, such as those given in Chapter 3. The ANHOF method was applied to a limited number of graphs that were from GCC version 2.58 compiler. These were chosen because they represented many areas of programming and there were many examples of varying size. The number and range of common model graphs is restrictive. The models may only be present in the GCC examples, there may be many more. It may be of benefit in improving the layout of the graphs if more and better graphs were found, say more fixed models. The layouts that were used to layout the common model graphs were chosen in order to improve the metrics and comprehension. It may be that other layouts may improve these criteria as well and therefore they should be tried. The size of graphs to which the ANHOF method has been applied is small. In software engineering there are many graphs of 1000 vertices or greater. For a true test of its ability then it should be applied to these as well.

A great deal of research has been carried out over the past few years into the representation of graphs. So that many aspects of the graph can be described in

languages. Several limited languages for representing the layout algorithms, the models and graph has been achieved in the ANHOF method. It may be of benefit if these were improved. This would increase the flexibility of the system. At the moment the layout algorithm languages requires a recompile of the Graph Layout System in order to implement a new algorithm. The use of a better language would improve this. The language used to represent the models is again complicated and fragmented because each type of model requires a different language. It may be of benefit to improve this language by combining it into one and also include the layout algorithms within it. Again the language used to represent the aesthetics does little more than allow the setting for the layout algorithms to be specified. Perhaps it may be beneficial to find a method in which to represent aesthetics better in a language.

9.7 Concluding Remarks

This chapter and thesis has shown that the ANHOF method of call graph layout, and its implementation the ANHOF system, has been very successful. It has shown that the method has met all its criteria for its success. It has discussed the diverse area of graph layout and shown that there are many different classes and uses of graphs. It has presented a successful method of graph layout for a specific type of graph, known as the call graph. The method uses subgraph isomorphism to search for common model graphs within these graphs. It has identified some common structures in call graphs and assigned a layout to them making them common model graphs. It has compared the layouts achieve using this method with layouts from other implementations of algorithms. It has compared this method with a similar theory by Kosak et al. [100] concluding that it has improved their method by solving problems identified with it. Finally it has suggested future work to be performed on the method and system. Overall it has shown that graph layout using subgraph isomorphisms is a successful method of graph layout.

10. Appendix 1 – Example Systems

10.1 Introduction

In Chapter 3 various graphs used in software engineering are given, together with various example graphs. In this Appendix a program and system are given that are used to layout the example graphs (Figure 19, Figure 21, Figure 23, and Figure 25)

10.2 Example Code

The following program provides an example in which the flowchart, call graph and control flow graphs can be drawn. It however has recursion present that means that the flowchart and control flow graph cannot accurately be drawn. The program is a relatively simple program for sorting. It is taken from section of Kernighan and Ritchie [93] and is called 'Lines.C'.

```
/* K & R pgs 108-110 */
#include <stdio.h>
#include <string.h>

#define MAXLINES 10 /* max #lines to be sorted */
#define MAXLEN  30 /* length of input line */
#define ALLOCSIZE 100 /* available space */

static char allocbuf[ALLOCSIZE];
static char *allocp = allocbuf;

char *lineptr[MAXLINES];
char *alloc(n)
int n;
{
 if (allocbuf + ALLOCSIZE - allocp >= n)
        {
        allocp += n;
         return allocp - n;
        }
 else
        return 0;
 }

int getline (s, lim)
 char s[];
 int lim;
{
 int c,i;
 i = 0;

 while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
```

```
        s[i++] = c;
 if (c == '\n')
        s[i++] = c;
 s[i] = '\0';
 return i;
}


int readlines(lineptr, maxlines)
char *lineptr[];
int maxlines;
{
 int len, nlines;
 char *p, line[MAXLEN];

 nlines = 0;
 while ((len = getline(line, MAXLEN)) > 0)
              {
              if (nlines >= maxlines)
                   return -1;
              if ((p =alloc(len)) == NULL)
                   return -1;
              line[len-1] = '\0';
              strcpy(p,line);
              lineptr[nlines++] = p;
              }
 return nlines;
}


writelines( lineptr, nlines)
char *lineptr[];
int nlines;
{
while (nlines-- > 0)
        printf('%s\n', *lineptr++);
}


swap(v, i, j)
char *v[];
int i,j;
{
         char *temp;
         temp = v[i];
         v[i] = v[j];
         v[j] = temp;
}


qsort(v, left, right)
char *v[];
int left, right;
{
 int i, last;
 if(left >= right)
        return;
 swap(v, left, (left+right)/2);
 last = left;
```

```
 for ( i=left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
                swap ( v, ++last, i);
 swap(v, left, last);
 qsort(v, left, last-1);
 qsort(v, last+1, right);
}


main()
{
int nlines;
 if ((nlines = readlines(lineptr, MAXLINES)) >= 0)
        {
        qsort(lineptr,0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
        }
 else
        {
        printf("error : input too big to sort\n");
        return 1;
        }
}
```

10.3 System Description

Gane [66] suggests that a data flow diagram can be drawn to model the following system. In Figure 25 this system has been drawn as such a diagram.

"Orders will be received by mail, or taken over the phone by the inward WATS line. Phone orders will be taken down in a standard form, or entered directly into a CRT using a standard format. Each order will be scanned to see that all important information is present, that the title exists (Or can be identified), and that the author is correct (Or can be identified), and that the book is available (i.e., not out of print). If the order is defective, it is routed to a supervisor to see if, e.g., "The Programming of Management," by Does Jane, should really be "The Management of Programming." by Jane Doe. Where payment is included, the amount is to be checked for correctness (if not correct, a request for further payment or a credit should be produced). Small discrepancies can be ignored. Where payment is not with the order, the customer file must be checked to see if the order comes from a person or organisation in good credit standing; if not, the person must be sent a confirmation of the order and a request for

prepayment. If the customer is new to us, an addition must be made to the customer file. For orders with payment or good credit, inventory is then to be checked to see if the order can be filled. If it can, a shipping note with an in voice (marked "paid" for prepaid orders) is prepared and sent out with the books. If the order can only be part-filled. A shipping note and invoice is prepared for the part shipment with a confirmation of the unfilled part (and paid invoice where payment was sent with the order), and a back order record is created. Back orders are to be filled as soon as the books are received from the publisher.

Where the order is for a book not held in inventory, the orders are batched for purchase requisition on the publisher when a quantity discount has been earned. Returned books are examined for damage, and entered back into stock, with a credit or refund being issued to the customer as appropriate. Where the returned book is not an inventory item, and the publisher allows returns, it is sent back to the publisher. When a shipment of books is received from a publisher, its contents are to be checked against the original purchase order, and discrepancies queried. The titles in the shipment are checked against the back orders for priority shipment, and the remainder entered into inventory. Inventory control policy calls for a reorder level on each title equal to the (average orders over the previous four weeks) X (delivery time from publishers) plus a 50 percent safety factor. Thus if sales of a title average 10 per week and the estimated delivery time is 3 weeks, an order will be placed with the publisher when the total copies in hand (and on order) have fallen to 45(3 × 10 × 150 percent). The safety factor may be varied from time to time by management, being increased for titles whose sales are rising and vice versa. The quantity for each order is determined by taking the product of the average order rate and delivery time, as above, multiplying by a bulk factor (normally 3), and rounding up to the next higher discount break point, unless that increases the order by more than 25 percent. Thus in the case above, the normal order would be 3 × 10 × 3 (bulk factor) or 90 copies. If the publisher offers an additional discount for orders of 100 or more, 100 would be ordered. If the discount is only offered for 120 or more, 90 would be ordered, since to order 120 would increase the order by the excessive amount of 33 percent. The bulk factor may be varied by management for each title from time to time. The calculation of average order rate includes not only orders that were filled, but frustrated demand, such as back orders, orders without payment,

and inquires that were not converted to orders because the book could not be supplied from stock.

When payments for books supplied are received, they are matched with the appropriate invoice. Where several invoices are outstanding for an account, and the payment does not match any one of them exactly, it is applied to the oldest invoice first. Frequently a customer will send one payment to cover several invoices. Where any invoice is more than 30 days overdue, a statement of all invoices outstanding is sent to the customer. When any invoice is more than 60 days overdue, a strongly worded letter is produced for the Vice President's signature.

When invoices are received from publishers, they are checked against the receipt-of-shipment records, and entered into accounts payable. If the discount for prompt payment given by the publisher exceeds, on an annualised bases, the marginal cost of funds (as specified from time to time by management), the system should produce a payment check on the last day the discount is available. For example, if 2.5 percent is offered for payment in 30 days, this is equivalent to 30 percent per year. The system should write a check on the 29th day.

Report of invoices sent out, by day, by week, by month, payments received by day, week, month, amounts overdue by various periods, stockouts, back orders, and purchases from publishers, should all be produced regularly. On demand analyses of sales by title, by subject, by publisher, with trend information, should be available on an immediate basis, together with information on publisher delivery times and purchasing trends. Immediate access to inventory figure of quantity-on-hand. quantity-on-order, and expected date of delivery are all very desirable, as is the facility to give a customer immediate information as to the status of his particular order. If a customer calls up and says, 'I sent you a cheque for £10 five weeks ago, for Bloggs book,' we would like to be able to tell him what day we shipped the book to him, or on what date we will be able to ship it."

11. Appendix 2 - File Formats

11.1 Introduction

In Chapter 3 many graph file formats are compared. Below are the grammars and example files of the four file formats that are compared. The call graph of the 'Lines.C' program in Appendix 1 is used to form an example of each file format.

11.2 daVinci

In Figure 80 the grammar forming a daVinci file is given and in Figure 81 an example file is given.

```
GRAPHTERM          ::= [GRAPHTERM1]

GRAPHTERM1         ::= NODE
                     | NODE,GRAPHTERM1

NODE               ::= l("STRING",n("",NODEATTRIBUTES,EDGES))
                     | r("STRING")

NODEATTRIBUTES     ::= [NODEATTRIBUTE]
                     | [NODEATTRIBUTE,NODEATTRIBUTES]

NODEATTRIBUTE      ::= a("OBJECT","STRING")
                     | a("FONTFAMILY","fontfamily")
                     | a("FONTSTYLE","fontstyle")
                     | a("COLOR","STRING")
                     | a("ICONFILE","STRING")
                     | a("_GO","BOXSTYLE")
                     | a("HIDDEN","BOOLEAN")
                     | a("BORDER","BORDERSTYLE")

fontfamily         ::= lucida
                     | times
                     | helvetica
                     | courier

fontstyle          ::= normal
                     | bold
                     | italic
                     | bold_italic

BOXSTYLE           ::= box
                     | circle
                     | ellispse
                     | rhombus
                     | text
                     | icon

BORDERSTYLE        ::= double
                     | single
```

```
EDGES              ::= [EDGE]
                     | [EDGE,EDGES]

EDGE               ::= l("STRING",e("",EDGEATTRIBUTES,NODE))

EDGEATTRIBUTES     ::= [EDGEATTRIBUTE]
                     | [EDGEATTRIBUTE,EDGEATTRIBUTES]

EDGEATTRIBUTE      ::= a("_DIR","DIRECTIONSTYLE")
                     | a("EDGEPATTERN","PATTERNSTYLE")
                     | a("EDGECOLOR","STRING");

DIRECTIONSTYLE     ::= normal
                     | inverse
                     | both
                     | none

PATTERNSTYLE       ::= solid
                     | dotted
                     | dashed
                     | thick
```

Figure 80 - The grammar definition of the daVinci language taken from [65]

The call graph of the 'Lines.C' program above can be represented as a daVinci file. This is shown below in Figure 81.

```
[l("main", n("", [a("OBJECT", "main")],
 [e("", [],
 l("printf", n("", [a("OBJECT", "printf")], []))),
 e("", [],
 l("qsort", n("", [a("OBJECT", "qsort")],
 [e("", [],
 r("qsort")),
 e("", [],
 l("strcmp", n("", [a("OBJECT", "strcmp")], []))),
 e("", [],
 l("swap", n("", [a("OBJECT", "swap")], []))),
 ]))),
 e("", [],
 l("readlines", n("", [a("OBJECT", "readlines")],
 [e("", [],
 l("getline", n("", [a("OBJECT", "getline")],
 [e("", [],
 l("getchar", n("", [a("OBJECT", "getchar")], []))),
 ]))),
 e("", [],
 l("alloc", n("", [a("OBJECT", "alloc")], []))),
 e("", [],
 l("strcpy", n("", [a("OBJECT", "strcpy")], []))),
 ]))),
 e("", [],
 l("writelines", n("", [a("OBJECT", "writelines")],
 [e("", [],
 r("printf")),
 ]))),
```

```
 ]))
]
```

Figure 81 - 'Lines.C' represented as a daVinci input file

11.3 Graph Tool

The Graph INformation (GIN) file format is much simpler than daVinci and in some ways more effective. In Figure 82, grammar of the file format is given and in Figure 83 an example file is given.

```
GRAPHTERM    ::= GRAPHTERM1 GRAPHTERM
               | GRAPHTERM1

GRAPHTERM1::= ( object ) INT INT INT INT INT ( TEXT ) ( TEXT )
               ( TEXT ) object
             | ( link ) INT INT INT INT INT INT INT INT ( TEXT )
               ( LINETYPE ) ( LINESTYLE ) link

TEXT         ::= STRING
               | _

LINETYPE     ::= directed

LINESTYLE    ::= LineSolid
```

Figure 82 - The grammar of a GIN file

The call graph of the 'Lines.C' program above can be represented as a Graph INformation (GIN) file. This is given in Figure 83.

```
( object ) 1 0 0 0 0 ( getline ) ( _ ) ( _ ) object
( object ) 2 0 0 0 0 ( getchar ) ( _ ) ( _ ) object
( link ) 1 2 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
( object ) 3 0 0 0 0 ( main ) ( _ ) ( _ ) object
( object ) 4 0 0 0 0 ( printf ) ( _ ) ( _ ) object
( link ) 3 4 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
( object ) 5 0 0 0 0 ( qsort ) ( _ ) ( _ ) object
( link ) 3 5 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
( object ) 6 0 0 0 0 ( readlines ) ( _ ) ( _ ) object
( link ) 3 6 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
( object ) 7 0 0 0 0 ( writelines ) ( _ ) ( _ ) object
( link ) 3 7 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
( link ) 5 5 0 0 0 0 0 0 ( 2 ) ( directed ) ( LineSolid ) link
( object ) 8 0 0 0 0 ( strcmp ) ( _ ) ( _ ) object
( link ) 5 8 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
( object ) 9 0 0 0 0 ( swap ) ( _ ) ( _ ) object
( link ) 5 9 0 0 0 0 0 0 ( 3 ) ( directed ) ( LineSolid ) link
( object ) 10 0 0 0 0 ( alloc ) ( _ ) ( _ ) object
( link ) 6 10 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
```

```
( link ) 6 1 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
( object ) 11 0 0 0 0 ( strcpy ) ( _ ) ( _ ) object
( link ) 6 11 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
( link ) 7 4 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
```

Figure 83 - The GIN file representation of 'Lines.C'

11.4 Graph Modelling Language

The Graph modelling language is a very powerful format because it is easily extended. In Figure 84 the grammar of the file format is given and an example file is given in Figure 85.

```
FILE              ::= LISTOFGRAPHS

LISTOFGRAPHS      ::= GRAPH
                    | GRAPH LISTOFGRAPHS

GRAPH             ::= graph [ GRAPHATTRIBUTES LISTOFNODES LISTOFEDGES]

GRAPHATTRIBUTES   ::= GRAPHATTRIBUTE
                    | GRAPHATTRIBUTE GRAPHATTRIBUTES

GRAPHATTRIBUTE    ::= GLOBALATTRIBUTE
                    | directed BOOLEAN
                    |
/*
Some attributes are common across the graph
*/


GLOBALATTRIBUTE   ::= id            INT
                    | label   STRING
                    | comment STRING
                    | Creator STRING
                    | name    STRING

/*
NODE Definition
*/


LISTOFNODES       ::= NODETYPE
                    | NODETYPE      LISTOFNODES
NODETYPE          ::= node [ NODEATTRIBUTES ]
                    |
NODEATTRUBUTES    ::= NODEATTRIBUTE
                    | NODEATTRIBUTE NODEATTRIBUTES
NODEATTRIBUTE     ::= edgeAnchor STRING
                    | GLOBALATTRIBUTE
                    | graphics [NODEGRAPHICATTS]
                    |
```

```
NODEGRAPHICATTS    ::= NODEGARPHICATT
                     | NODEGRAPHICATT NODEGRAPHICATTS

NODEGRAPHICATT     ::= x              REAL        //x coord
                     | y              REAL        //y coord
                     | z              REAL        //z coord
                     | w              REAL        //width
                     | h              REAL        //height
                     | d              REAL        //height
                     | type           TYPEVALUE   //type of graphic
                     | image          STRING
                                           //name of filecontaining image
                                      //note scales to width and height
                     | bitmap         STRING
                                      //file that is just black and white
                     |

TYPEVALUE          ::= "arc"
                     | "bitmap"
                     | "image"
                     | "line"
                     | "oval"
                     | "polygon"
                     | "rectangle"
                     | "text"

/*
EDGE Definition
*/

LISTOFEDGES        ::= EDGETYPE
                     | EDGETYPE LISTOFEDGES

EDGETYPE           ::= edge [ EDGEATTRIBUTES ]
                     |
EDGEATTRIBUTES     ::= EDGEATTRIBUTE
                     | EDGEATTRIBUTE EDGEATTRIBUTES

EDGEATTRIBUTE      ::= source INT                // Edge from node id
                     | target INT                // Edge to node id
                     | GLOBALATTRIBUTE
                     | graphics [EDGEGRAPHICATTS]
                     |

EDGEGRAPHICATTS    ::= EDGEGRAPHICATT
                     | EDGEGRAPHICATT EDGEGRAPHICATTS

EDGEGRAPHICATT     ::= NODEGRAPHICATT
                     | width REAL          //thickness of line
                     | stripple STRING     //type of line file
                     | Line [ POINTLIST ]
/*
A line is described in terms of two endpoints and a list of bends.
A bend and endpoint are just normal point in the Cartesian plane
*/
POINTLIST          ::= POINT
                     | POINT POINTLIST

POINT              ::= point [ LINEATTRIBUTES ]
```

```
LINEATTRIBUTES    ::= LINEATTRIBUTE
                    |  LINEATTRIBUTE LINEATTRIBUTES

LINEATTRIBUTE     ::= x REAL
                    |  y REAL
                    |  z REAL
                    |
```

Figure 84 - The grammar of a GML file taken from [79]

Figure 85 shows the program of 'Lines.C' represented as a Graph Modelling Language (GML) file.

```
graph [

comment "This is the gml version of the call graph of Lines.C"
directed 1
isplanar 0
node  [
      id 1
      label "getline"
      ]

node  [
      id 2
      label "getchar"
      ]

node  [
      id 3
      label "main"
      ]

node  [
      id 4
      label "printf"
      ]

node  [
      id 5
      label "qsort"
      ]

node  [
      id 6
      label "readlines"
      ]

node  [
      id 7
      label "writelines"
      ]
```

```
node   [
       id 8
       label "strcmp"
       ]

node   [
       id 9
       label "swap"
       ]

node   [
       id 10
       label "alloc"
       ]
node   [
       id 11
       label "strcpy"
       ]

edge   [
       source 1
       target 2
       label "1"
       ]

edge   [
       source 3
       target 4
       label "1"
       ]

edge   [
       source 3
       target 5
       label "1"
       ]

edge   [
       source 3
       target 6
       label "1"
       ]

edge   [
       source 3
       target 7
       label "1"
       ]

edge   [
       source 5
       target 5
       label "2"
       ]

edge   [
       source 5
       target 8
       label "1"
       ]
```

```
edge  [
      source 5
      target 9
      label "3"
      ]

edge  [
      source 6
      target 1
      label "1"
      ]

edge  [
      source 6
      target 10
      label "1"
      ]

edge  [
      source 6
      target 11
      label "1"
      ]

edge  [
      source 7
      target 4
      label "1"
      ]
]
```

Figure 85 - 'Lines.C' represented as a GML file

11.5 VCG

Again the VCG file format is powerful performing many features the others do not. In Figure 86 the grammar for the file format is given and in Figure 87 an example file is given.

```
graph       ::= "graph:" '{' graph_entry_list '}'
              ;
graph_entry_list::= graph_entry_list graph_entry
              | graph_entry
              ;
graph_entry ::= graph_attribute
              | node_defaults
              | edge_defaults
              | foldnode_defaults
              | foldedge_defaults
              | graph
              | node
              | edge
              | nearedge
              ;
```

```
graph_attribute   ::= 'x' ':' integer
                    | 'y' ':' integer
                    | "loc:" '{' 'x' ':' integer 'y' ':' int_const '}'
                    | "width" ':' integer
                    | "height" ':' integer
                    | "xmax" ':' integer
                    | "ymax" ':' integer
                    | "xbase" ':' integer
                    | "ybase" ':' integer
                    | "xspace" ':' integer
                    | "xlspace" ':' integer
                    | "yspace" ':' integer
                    | "xraster" ':' integer
                    | "xlraster" ':' integer
                    | "yraster" ':' integer
                    | "folding" ':' integer
                    | "invisible" ':' integer
                    | "hidden" ':' integer
                    | "title" ':' string
                    | "label" ':' string
                    | "classname" integer ':' string
                    | "infoname" integer ':' string
                    | "info1" ':' string
                    | "info2" ':' string
                    | "info3" ':' string
                    | "textmode" ':' enum_textmode
                    | "borderwidth" ':' integer
                    | "color" ':' enum_color
                    | "textcolor" ':'enum_color
                    | "bordercolor" ':'enum_color
                    | "orientation" ':' enum_orientation
                    | "node_alignment" ':' enum_node_align
                    | "scaling" ':' float
                    | "shrink" ':' integer
                    | "stretch" ':' integer
                    | "layoutalgorithm" ':' enum_layoutalgorithm
                    | "layout_downfactor" ':' integer
                    | "layout_upfactor" ':' integer
                    | "layout_nearfactor" ':' integer
                    | "layout_splinefactor" ':' integer
                    | "splinefactor" ':' integer
                    | "status" ':' enum_status
                    | "late_edge_labels" ':' enum_yes_no
                    | "display_edge_labels" ':' enum_yes_no
                    | "dirty_edge_labels" ':' enum_yes_no
                    | "finetuning" ':' enum_yes_no
                    | "splines" ':'    enum_yes_no
                    | "no_nearedges"
                    | "nearedges" ':' "no"
                    | "nearedges" ':' "yes"
                    | "shape" ':' enum_shape
                    | "level" ':' integer
                    | "vertical_order" ':' integer
                    | "horizontal_order" ':' integer
                    | "crossing_optimization" ':' enum_yes_no
                    | "crossing_weight" ':' enum_cross_weight
                    | "spreadlevel"     ':' integer
                    | "treefactor" ':' float
                    ;
enum_color  ::= "aquamarine"
```

```
                | "black"
                | "blue"
                | "cyan"
                | "darkblue"
                | "darkcyan"
                | "darkgreen"
                | "darkgrey"
                | "darkmagenta"
                | "darkred"
                | "darkyellow"
                | "gold"
                | "green"
                | "khaki"
                | "lightblue"
                | "lightcyan"
                | "lightgreen"
                | "lightgrey"
                | "lightmagenta"
                | "lightred"
                | "lightyellow"
                | "lilac"
                | "magenta"
                | "orange"
                | "orchid"
                | "pink"
                | "purple"
                | "red"
                | "turquoise"
                | "white"
                | "yellow"
                | "yellowgreen"
                ;
enum_orientation::= "top_to_bottom"
                | "bottom_to_top"
                | "left_to_right"
                | "right_to_left"
                ;
enum_layoutalgorithm::=
                | "tree"
                | "maxdepth"
                | "mindepth"
                | "maxdepthslow"
                | "mindepthslow"
                | "maxdegree"
                | "mindegree"
                | "maxindegree"
                | "minindegree"
                | "maxoutdegree"
                | "minoutdegree"
                | "minbackward"
                ;
enum_status       ::= "black"
                | "grey"
                | "white"
                ;
enum_yes_no ::= "yes"
                | "no";
enum_cross_weight ::= "bary"
                  | "median";
foldnode_defaults::= "foldnode." node_attribute;
```

```
foldedge_defaults::= "foldedge." edge_attribute;
node_defaults     ::= "node." node_attribute;
edge_defaults   ::= "edge." edge_attribute  ;
node        ::= "node:" '{' node_attribute_list '}';
node_attribute_list::= node_attribute_list node_attribute
                | node_attribute;
edge            ::= "edge:" '{' edge_attribute_list '}';
nearedge    ::= "nearedge:" '{' edge_attribute_list '}';
edge_attribute_list::= edge_attribute_list edge_attribute
                    | edge_attribute;
node_attribute    ::= "title" ':' string
              | "label" ':' string
              | "info1" ':' string
              | "info2" ':' string
              | "info3" ':' string
                | "color" ':' enum_color
              | "textcolor" ':'enum_color
                | "bordercolor" ':'enum_color
              | "width" ':' integer
              | "height" ':' integer
                | "borderwidth" ':' integer
              | "loc:" '{' 'x' ':' integer 'y' ':' int_const '}'
              | "folding" ':' integer
              | "scaling" ':' float
              | "shrink" ':' integer
              | "stretch" ':' integer
              | "textmode" ':' enum_textmode
              | "shape" ':' enum_shape
              | "level" ':' integer
              | "vertical_order" ':' integer
              | "horizontal_order" ':' integer
              ;
enum_textmode     ::= "center"
              | "left_justify"
              | "right_justify"
              ;
enum_shape  ::= "box"
              | "rhomb"
              | "ellipse"
              | "triangle"
              ;
enum_node_align   ::= "bottom"
              | "top"
              | "center"
              ;
edge_attribute  ::= "sourcename" ':' string
              | "targetname" ':' string
                | "label" ':' string
                | "color" ':' enum_color
              | "thickness" ':' integer
              | "class" ':' integer
              | "priority" ':' integer
              | "arrowsize" ':' integer
              | "linestyle" ':' enum_linestyle
              | "anchor" ':' integer
                | "horizontal_order" ':' integer
              ;
enum_linestyle    ::= "continuous"
              | "solid"
              | "dotted"
```

```
| "dashed"
| "invisible"
;
```

Figure 86- The input grammar of a VCG file taken from [103]

Figure 87 shows how the program of 'Lines.C' can be represented as a VCG input file.

```
graph: {

xspace:  25

node: {title:"1" label:"getline" loc:{x:0 y:0}}
node: {title:"2" label:"getchar" loc:{x:0 y:0}}
node: {title:"3" label:"main" loc:{x:0 y:0}}
node: {title:"4" label:"printf" loc:{x:0 y:0}}
node: {title:"5" label:"qsort" loc:{x:0 y:0}}
node: {title:"6" label:"readlines" loc:{x:0 y:0}}
node: {title:"7" label:"writelines" loc:{x:0 y:0}}
node: {title:"8" label:"strcmp" loc:{x:0 y:0}}
node: {title:"9" label:"swap" loc:{x:0 y:0}}
node: {title:"10" label:"alloc" loc:{x:0 y:0}}
node: {title:"11" label:"strcpy" loc:{x:0 y:0}}
edge: { thickness: 3 sourcename :"1" targetname: "2" label: "1" }
edge: { thickness: 3 sourcename :"3" targetname: "4" label: "1" }
edge: { thickness: 3 sourcename :"3" targetname: "5" label: "1" }
edge: { thickness: 3 sourcename :"3" targetname: "6" label: "1" }
edge: { thickness: 3 sourcename :"3" targetname: "7" label: "1" }
edge: { thickness: 3 sourcename :"5" targetname: "5" label: "2" }
edge: { thickness: 3 sourcename :"5" targetname: "8" label: "1" }
edge: { thickness: 3 sourcename :"5" targetname: "9" label: "3" }
edge: { thickness: 3 sourcename :"6" targetname: "1" label: "1" }
edge: { thickness: 3 sourcename :"6" targetname: "10" label: "1" }
edge: { thickness: 3 sourcename :"6" targetname: "11" label: "1" }
edge: { thickness: 3 sourcename :"7" targetname: "4" label: "1" }

}
```

Figure 87 - 'Lines.C' represented as a VCG input file

12. Appendix 3 – Implementation Information

When using the ANHOF system various input files are needed and output files are created. The Graph Isomorphism System uses various fact bases that are used by the Prolog programs to detect the variable models in the Variable Model Detection System. These fact bases are discussed below. Chapter 4 suggests a method of describing new variable models using a language. In Chapter 5 it is shown that the language in the ANHOF system that describes the models is that of Prolog Rules. Prolog is a complicated language to learn, in order to ease this process various output routines have been written that ease the output of the fact bases in the new variable model descriptions, these are given below. The Match Analyser produces a new representation of the graph which details all the common model graphs present in the graph. Detailed below is the grammar for the representation and also Prolog routines for its output.

12.1 Fact Bases

One part of the Graph Isomorphism System detects variable models. This system is programmed in Prolog, a program that uses logic to process fact bases. In order to detect models graphs two fact bases are used, the graph fact base and the fan information fact base. All parts of the Graph Isomorphism System output matches to the models valid or invalid for use elsewhere. These fact bases are detailed below.

12.1.1 Graph Fact Base

This fact base provides information on the vertices and edges of a graph. In terms of the Graph Isomorphism System and Match Analyser this is the graph. All information about the graph must be contained in here. The eventual graph display system is Graph Tool this allowing three lines of text for each vertex. Also vertices are allowed to be coloured, the colouring scheme allowed by graph tool for colouring both vertices and edges is red, green, orange, black, white, magenta, purple and blue. Each vertex is given a unique identifier and a coordinate. This information should be reflected in the vertex (node) fact given below in Definition 6.

node(Id,X,Y,Text1,Text2,Text3,Colour).

Where Id is an integer and is a unique identifier that cannot equal –1.

X & Y are the members of the coordinate (X, Y) and are integers.

Text1, Text2 and Text3 is a string and are the lines of text that can be present in the label for a vertex.

Colour is a string and can be one of "RED", "GREEN", "ORANGE", "WHITE", "BLACK", "MAGENTA", "BLUE", "YELLOW" or "PURPLE".

Definition 6- A node fact

A call graph is a directed graph; therefore an edge has a vertex to go to and from. Each edge can be labelled and can have a different line style either dashed or solid, however this is not implemented in later versions of Graph Tool. An edge can be directed, bi directed, reverse directed or undirected in Graph Tool. In a call graph edges cannot be bi directed or undirected and the use of reverse directed should be avoided. Again an edge can be given a colour. This information should be encapsulated in an edge fact given below in Definition 7.

edge(From,To,Text,Linetype,Linestyle,Colour).

Where:

From & To are integers and are the identifiers of the vertices that the edge goes from and to, there must be a node fact with the same integer.

Text is of string type and is the label attached to the edge.

Linetype is a string type and can be one of "directed", "bidirected", "reverse_directed" or "undirected".

Linestyle is a string and is the style of the line and can only be "LineSolid" to work with the latest versions of graph tool.

Colour is a string and can be one of "RED", "GREEN", "ORANGE", "WHITE", "BLACK", "MAGENTA", "BLUE", "YELLOW" or "PURPLE".

Definition 7 - An edge fact

12.1.2 Fan Information Fact Base

This is created by the fan in and fan out information calculator. It creates a list of "fan facts" which contain the number of edges coming into a vertex (fan in) and the number of edges leaving a vertex (fan out). There must be a fan fact for each vertex. It is used in the detection of variable models. A fan fact is given below in Definition 8.

fan(Nodeid,Fanin,Fanout).

Where Nodeid is an integer that is the identifier of the vertex in question, it should match the id of a node fact.

Fanout & Fanin are integers that represent the fan information.

Definition 8 - A fan fact

12.1.3 Match Fact Base

The Graph Isomorphism System produces a list of matches to the models that it has searched for. The list of matches is in the form of a Prolog fact base. This enables the Match Analyser to process them and remove any invalid matches; the valid matches are also outputted as a Prolog fact base. These fact bases are of the same format and are known as the Match Fact Base. A match consists of a model name and a list of the vertices that are members of the model. A match fact contains this information and is given below in Definition 9.

match(Modelname,Listofnodesinvolved).

Where Modelname is a string and is the name of the model found

Listofnodesinvolved is a list of integer values that are the identifiers of the vertices involved.

Definition 9 - A match fact

12.2 The Graph Representation

Given below in Definition 10 is the language definition for the graph representation. The graph representation is meant to be a simple English representation of a graph showing the valid models that are present in that graph.

The representation consists of two parts, the graph part and the structures part. The graph part is just a list of vertices and edges involved in the whole graph. The structures part is where the models are given. The information contained in a node and edge fact is same information that is stored in the graph part. Also as well as the coordinate of the vertices, the vertex height and width can also be obtained from the representation. This is obtained by using the coordinates of the top left and bottom right corners of the vertex. Also contained in the graph part is the name of the algorithm that will be used to lay out the main graph. The structures part of the representation is used to represent a model, giving the model name and vertices involved. Also the name of the algorithm to lay out the model.

```
REPRESENTATION ::= GRAPH STRUCTURES

GRAPH ::= graph {NODES EDGES ALGORITHM}

NODES ::= [NODE]
NODE                                                    ::=
node(ID,X1,Y1,X2,Y2,TEXT1,TEXT2,TEXT3,COLOUR).

ID ::= INTEGER                                          // unique identifier
X1 ::= INTEGER                                          // top left x coord
Y1 ::= INTEGER                                          // top left y coord
X2 ::= INTEGER                                          // bottom right x coord
Y2 ::= INTEGER                                          // bottom right y coord
TEXT1 ::= STRING                                        // 1st line of text
TEXT1 ::= STRING                                        // 2nd line of text
TEXT1 ::= STRING                                        // 3rd line of text
COLOUR ::= "RED"
         | "GREEN"
         | "ORANGE"
         | "WHITE"
         | "BLACK"
         | "MAGENTA"
         | "BLUE"
         | "YELLOW"
         | "PURPLE"
```

```
EDGES ::= [EDGE]
EDGE                                                    ::=
edge(FROM,TO,TEXT,LINETYPE,LINESTYLE,COLOUR).

FROM ::= INTEGER                                        // From Vertex ID
TO ::= INTEGER                                          // To Vertex ID
TEXT ::= STRING                                         // edge label
LINETYPE ::= "directed"                                 // Type of line
           | "bidirected"
           | "reverse_directed"
           | "undirected"
LINESTYLE ::= "LineSolid"                               // Syle of line
ALGORITHM ::= algorithm(STRING)                         // name of layout
                                                        // algorithm to use

STRUCTURES ::= [STRUCTURE]
STRUCTURE  ::=  structure{MODELNAME  ALGORITHM
NODESUSED}

MODELNAME ::= name(STRING).

NODESUSED ::= [NODEUSED]
NODEUSED ::= nodeused(INTEGER).
```

Definition 10 - The Graph Representation

12.3 Prolog Rule Output Routines

Detailed below are the routines that are used to aid writing new rules to represent variable models. They are used to output the matches to the common model graph being described. So that the Match Analyser can process them. The routines are also used to output the valid matches from the Match Analyser. Prolog routines that output the representation from the Match Analyser are also detailed.

12.3.1 Match Fact Base

The Match Fact Base is used to output valid and invalid matches to the common model graphs, from the Variable Model Detection System. They are used to output the valid matches from the Match Analyser.

12.3.1.1 Write Match Rule

This routine outputs a match to a common model graph in the match fact base format. It takes as an input parameter a model name (*Modelname*) in the form of a string and a list

of integers (*Listofvertices*) that represent the identification numbers of the vertices involved.

It should be used in Prolog in the following format: -

writematch(*Modelname,Listofvertices*).

12.3.2 Graph Representation File

The graph representation is used to represent the graph and the common model graphs present in the graph to the Graph Layout System. The Match Analyser creates it, which is a Prolog based program. Detailed below is the rules used in the Prolog program to create the file.

12.3.2.1 Write Node Rule

This outputs a node (vertex) fact to the representation file. It is used so that the whole graph is in the representation file. It takes all the information about a vertex its id number, its x and y coordinate and any text and colour and outputs it to the representation file. The colour can only be one of "RED", "GREEN", "ORANGE", "WHITE, "BLACK", "MAGENTA", "BLUE", "YELLOW" or "PURPLE".

It should be used in Prolog in the following format: -

writenode(*Id,X,Y,Text1,Text2,Text3,Colour*).

12.3.2.2 Write Nodes Rule

This outputs all the nodes (vertices) that are currently stored to the representation file.

It should be used in Prolog in the following format: -

writenodes(_).

12.3.2.3 Write Edge Rule

This outputs an edge fact to the representation file. It is again used so that the original graph is in the representation file. It takes all the information about an edge, which vertex identification numbers it goes to and form, the labels, line style and type and the colour. The line style can only be "LineSolid" to work with the latest version of Graph Tool. The colour can only be one of "RED", "GREEN", "ORANGE", "WHITE, "BLACK", "MAGENTA", "BLUE", "YELLOW" or "PURPLE".

It should be used in Prolog in the following format: -

writeedge(*From,To,Label,Linetype,Linestyle,Colour*).

12.3.2.4 Write Edges Rule

This outputs all the edges to the representation file that are currently stored.

It should be used in Prolog in the following format: -

writeedges(_).

12.3.2.5 Write Graph Rule

This outputs the whole graph in format used in the representation file. It gets its information from the node and edge facts that are currently stored from the graph fact base.

It should be used in Prolog in the following format: -

writegraph(_).

12.3.2.6 Write Structure Rule

This outputs a valid match to the representation file. It informs the Graph Layout System that a graph has a certain (*Modelname*) structure present and it involves the vertices with the vertex identification numbers contained *Listofvertices.*

writestructure(*Modelname,Listofvertices*)

13. Appendix 4 – The ANHOF System at Work

In Chapter 5 the implementation of the ANHOF method is discussed. The implementation is known as the ANHOF system and architecture of the system is shown in Figure 43. This Appendix details the input and output files for each program in order to represent and layout Graph G in Chapter 7.

13.1 Adjacency Matrices

When detecting the fixed common model graphs Messmer's Graph Matching Toolkit searches for the adjacency matrix in the graph, in order to perform this the adjacency matrix has to be fed in. Given below is the adjacency matrix of Triangle and Box common model graph and the GIN input file for them so that they can be used to detect them.

13.1.1 Triangle Common Model Graph

The adjacency matrix of a Triangle common model graph is given in Table 22.

	A	B	C
A	0	1	1
B	0	0	1
C	0	0	0

Table 22 - The adjacency matrix of a Triangle common model graph

Table 22 can be represented as a GIN file given in Figure 88.

(object) 1 0 0 0 0 (A) (_) (_) (_) (_) object
(object) 2 0 0 0 0 (B) (_) (_) (_) (_) object
(object) 3 0 0 0 0 (C) (_) (_) (_) (_) object
(link) 1 2 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 2 3 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 1 3 0 0 0 0 0 0 (_) (directed) (LineSolid) link

Figure 88 – The GIN representation of a Triangle common model graph

13.1.2 Box Common Model Graph

A Box common model graph can be represented as an adjacency matrix, it is given in Table 23.

	A	B	C	D
A	0	1	0	0
B	0	0	1	0
C	0	0	0	0
D	1	0	1	0

Table 23 - The adjacency of a Box common model graph

Table 23 can be represented as a GIN file in the following manner: -

(object) 1 0 0 0 0 (A) (_) (_) object
(object) 2 0 0 0 0 (B) (_) (_) object
(object) 3 0 0 0 0 (C) (_) (_) object
(object) 4 0 0 0 0 (D) (_) (_) object
(link) 1 2 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 2 3 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 4 1 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 4 3 0 0 0 0 0 0 (_) (directed) (LineSolid) link

Figure 89 - The GIN representation of a Box common model graph

13.2 GIN Input file

The ANHOF system takes a graph description in the form of a Graph INformation (GIN) file. Details of which are given in Chapter 3 and Appendix 2. The file format is a list of vertices and edges. The GIN input file is given below in Figure 90.

```
( object ) 1 0 0 0 0 ( 1 ) ( _ ) ( _ ) object
( object ) 2 0 0 0 0 ( 2 ) ( _ ) ( _ ) object
( object ) 3 0 0 0 0 ( 3 ) ( _ ) ( _ ) object
( object ) 4 0 0 0 0 ( 4 ) ( _ ) ( _ ) object
( object ) 5 0 0 0 0 ( 5 ) ( _ ) ( _ ) object
( object ) 6 0 0 0 0 ( 6 ) ( _ ) ( _ ) object
( object ) 7 0 0 0 0 ( 7 ) ( _ ) ( _ ) object
( object ) 8 0 0 0 0 ( 8 ) ( _ ) ( _ ) object
( object ) 9 0 0 0 0 ( 9 ) ( _ ) ( _ ) object
( object ) 10 0 0 0 0 ( 10 ) ( _ ) ( _ ) object
( object ) 11 0 0 0 0 ( 11 ) ( _ ) ( _ ) object
( object ) 12 0 0 0 0 ( 12 ) ( _ ) ( _ ) object
( object ) 13 0 0 0 0 ( 13 ) ( _ ) ( _ ) object
( object ) 14 0 0 0 0 ( 14 ) ( _ ) ( _ ) object
( object ) 15 0 0 0 0 ( 15 ) ( _ ) ( _ ) object
( object ) 16 0 0 0 0 ( 16 ) ( _ ) ( _ ) object
( object ) 17 0 0 0 0 ( 17 ) ( _ ) ( _ ) object
( object ) 18 0 0 0 0 ( 18 ) ( _ ) ( _ ) object
( object ) 19 0 0 0 0 ( 19 ) ( _ ) ( _ ) object
( object ) 20 0 0 0 0 ( 20 ) ( _ ) ( _ ) object
( object ) 21 0 0 0 0 ( 21 ) ( _ ) ( _ ) object
( object ) 22 0 0 0 0 ( 22 ) ( _ ) ( _ ) object
( object ) 23 0 0 0 0 ( 23 ) ( _ ) ( _ ) object
( object ) 24 0 0 0 0 ( 24 ) ( _ ) ( _ ) object
( object ) 25 0 0 0 0 ( 25 ) ( _ ) ( _ ) object
( object ) 26 0 0 0 0 ( 26 ) ( _ ) ( _ ) object
( object ) 27 0 0 0 0 ( 27 ) ( _ ) ( _ ) object
( object ) 28 0 0 0 0 ( 28 ) ( _ ) ( _ ) object
( object ) 29 0 0 0 0 ( 29 ) ( _ ) ( _ ) object
( object ) 30 0 0 0 0 ( 30 ) ( _ ) ( _ ) object
( object ) 31 0 0 0 0 ( 31 ) ( _ ) ( _ ) object
( object ) 32 0 0 0 0 ( 32 ) ( _ ) ( _ ) object
( object ) 33 0 0 0 0 ( 33 ) ( _ ) ( _ ) object
( object ) 34 0 0 0 0 ( 34 ) ( _ ) ( _ ) object
( object ) 35 0 0 0 0 ( 35 ) ( _ ) ( _ ) object
( object ) 36 0 0 0 0 ( 36 ) ( _ ) ( _ ) object
( object ) 37 0 0 0 0 ( 37 ) ( _ ) ( _ ) object
( object ) 38 0 0 0 0 ( 38 ) ( _ ) ( _ ) object
( object ) 39 0 0 0 0 ( 39 ) ( _ ) ( _ ) object
( object ) 40 0 0 0 0 ( 40 ) ( _ ) ( _ ) object
( object ) 41 0 0 0 0 ( 41 ) ( _ ) ( _ ) object
( object ) 42 0 0 0 0 ( 42 ) ( _ ) ( _ ) object
( object ) 43 0 0 0 0 ( 43 ) ( _ ) ( _ ) object
( object ) 44 0 0 0 0 ( 44 ) ( _ ) ( _ ) object
( object ) 45 0 0 0 0 ( 45 ) ( _ ) ( _ ) object
( link ) 1 18 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 1 2 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 1 37 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 1 41 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 1 6 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 10 11 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 10 12 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 10 13 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 14 15 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 16 14 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 17 15 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 17 16 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 18 19 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 19 20 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 2 3 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
```

```
( link ) 20 21 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 20 22 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 20 23 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 21 14 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 22 24 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 23 32 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 24 25 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 24 26 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 24 27 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 28 25 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 28 26 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 28 27 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 28 29 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 28 30 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 28 31 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 3 4 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 32 33 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 32 34 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 32 35 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 36 33 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 36 34 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 36 35 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 37 38 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 38 39 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 38 40 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 4 5 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 40 39 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 41 42 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 43 42 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 44 42 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 45 42 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 6 7 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 7 10 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 8 10 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 9 10 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
```

Figure 90 - The GIN input file representing graph G

13.3 Graph Isomorphism System

In Chapter 4 it was shown that there were 2 types of models in Graphs, Variable and Fixed. Chapter 5 showed how this would be implemented in 2 detection methods; one that detects fixed models based around Messmer's [114] work and another detecting variable models based on a Prolog program. The following section will show both at work.

13.3.1 Variable Model Isomorphism

There are a higher proportion of variable models to detect in a call graph. These are models that can involve any number of vertices. In order for the Prolog system to work the graph G given in Chapter 7 needs to be converted into Prolog facts.
Figure 91 shows the Prolog representation of the graph.

```
node(1,0,0,"1","_","_","Black").
node(2,0,0,"2","_","_","Black").
node(3,0,0,"3","_","_","Black").
node(4,0,0,"4","_","_","Black").
node(5,0,0,"5","_","_","Black").
node(6,0,0,"6","_","_","Black").
node(7,0,0,"7","_","_","Black").
node(8,0,0,"8","_","_","Black").
node(9,0,0,"9","_","_","Black").
node(10,0,0,"10","_","_","Black").
node(11,0,0,"11","_","_","Black").
node(12,0,0,"12","_","_","Black").
node(13,0,0,"13","_","_","Black").
node(14,0,0,"14","_","_","Black").
node(15,0,0,"15","_","_","Black").
node(16,0,0,"16","_","_","Black").
node(17,0,0,"17","_","_","Black").
node(18,0,0,"18","_","_","Black").
node(19,0,0,"19","_","_","Black").
node(20,0,0,"20","_","_","Black").
node(21,0,0,"21","_","_","Black").
node(22,0,0,"22","_","_","Black").
node(23,0,0,"23","_","_","Black").
node(24,0,0,"24","_","_","Black").
node(25,0,0,"25","_","_","Black").
node(26,0,0,"26","_","_","Black").
node(27,0,0,"27","_","_","Black").
node(28,0,0,"28","_","_","Black")
node(29,0,0,"29","_","_","Black")
node(30,0,0,"30","_","_","Black")
node(31,0,0,"31","_","_","Black")
node(32,0,0,"32","_","_","Black").
node(33,0,0,"33","_","_","Black").
node(34,0,0,"34","_","_","Black").
node(35,0,0,"35","_","_","Black").
node(36,0,0,"36","_","_","Black").
node(37,0,0,"37","_","_","Black").
node(38,0,0,"38","_","_","Black").
node(39,0,0,"39","_","_","Black").
node(40,0,0,"40","_","_","Black").
```

```
node(41,0,0,"41","_","_","Black").
node(42,0,0,"42","_","_","Black").
node(43,0,0,"43","_","_","Black").
node(44,0,0,"44","_","_","Black").
node(45,0,0,"45","_","_","Black").
edge(1,18,"_","directed","LineSolid","Black").
edge(1,2,"_","directed","LineSolid","Black").
edge(1,37,"_","directed","LineSolid","Black").
edge(1,41,"_","directed","LineSolid","Black").
edge(1,6,"_","directed","LineSolid","Black").
edge(2,3,"_","directed","LineSolid","Black").
edge(3,4,"_","directed","LineSolid","Black").
edge(4,5,"_","directed","LineSolid","Black").
edge(6,7,"_","directed","LineSolid","Black").
edge(7,10,"_","directed","LineSolid","Black").
edge(8,10,"_","directed","LineSolid","Black").
edge(9,10,"_","directed","LineSolid","Black").
edge(10,11,"_","directed","LineSolid","Black").
edge(10,12,"_","directed","LineSolid","Black").
edge(10,13,"_","directed","LineSolid","Black").
edge(14,15,"_","directed","LineSolid","Black").
edge(17,15,"_","directed","LineSolid","Black").
edge(14,16,"_","directed","LineSolid","Black").
edge(17,16,"_","directed","LineSolid","Black").
edge(18,19,"_","directed","LineSolid","Black").
edge(19,20,"_","directed","LineSolid","Black").
edge(20,21,"_","directed","LineSolid","Black").
edge(20,22,"_","directed","LineSolid","Black").
edge(20,23,"_","directed","LineSolid","Black").
edge(21,14,"_","directed","LineSolid","Black").
edge(22,24,"_","directed","LineSolid","Black").
edge(23,32,"_","directed","LineSolid","Black").
edge(24,25,"_","directed","LineSolid","Black").
edge(24,26,"_","directed","LineSolid","Black").
edge(24,27,"_","directed","LineSolid","Black").
edge(28,25,"_","directed","LineSolid","Black").
edge(28,26,"_","directed","LineSolid","Black").
edge(28,27,"_","directed","LineSolid","Black").
edge(28,29,"_","directed","LineSolid","Black").
edge(28,30,"_","directed","LineSolid","Black").
edge(28,31,"_","directed","LineSolid","Black").
edge(32,33,"_","directed","LineSolid","Black").
edge(32,34,"_","directed","LineSolid","Black").
edge(32,35,"_","directed","LineSolid","Black").
edge(36,33,"_","directed","LineSolid","Black").
edge(36,34,"_","directed","LineSolid","Black").
edge(36,35,"_","directed","LineSolid","Black").
edge(37,38,"_","directed","LineSolid","Black").
edge(38,39,"_","directed","LineSolid","Black").
```

```
edge(38,40,"_","directed","LineSolid","Black").
edge(40,39,"_","directed","LineSolid","Black").
edge(41,42,"_","directed","LineSolid","Black").
edge(43,42,"_","directed","LineSolid","Black").
edge(44,42,"_","directed","LineSolid","Black").
edge(45,42,"_","directed","LineSolid","Black")
```

Figure 91 - The Prolog representation of Graph G

This graph then has to be processed for the fan in and fan out information. This simply is the numbers of vertices coming to and from a given node. The flow information for graph G is given in
Figure 92.

```
fan(1,0,5).
fan(2,1,1).
fan(3,1,1).
fan(4,1,1).
fan(5,1,0).
fan(6,1,1).
fan(7,1,1).
fan(8,0,1).
fan(9,0,1).
fan(10,3,3).
fan(11,1,0).
fan(12,1,0).
fan(13,1,0).
fan(14,2,1).
fan(15,2,0).
fan(16,1,1).
fan(17,0,2).
fan(18,1,1).
fan(19,1,1).
fan(20,1,3).
fan(21,1,1).
fan(22,1,1).
fan(23,1,1).
fan(24,1,3).
fan(25,2,0).
fan(26,2,0).
fan(27,2,0).
fan(28,0,6).
fan(29,1,0).
fan(30,1,0).
fan(31,1,0).
fan(32,1,3).
fan(33,2,0).
```

```
fan(34,2,0).
fan(35,2,0).
fan(36,0,3).
fan(37,1,1).
fan(38,1,2).
fan(39,2,0).
fan(40,1,1).
fan(41,1,1).
fan(42,4,0).
fan(43,0,1).
fan(44,0,1).
fan(45,0,1).
```

Figure 92 - fan in and fan out information

The various Prolog programs search the fan in and fan out information for the relevant information. Details of the algorithms are given in Chapter 4 and 6.

13.3.1.1 Fan In Models

The Variable Model Detection System found that vertices 10 and 42 are fan in vertices. The vertices that are involved in the models are 9,8,7,41,43,44,45. The output from the Fan In model search part of the Variable Model Detection System is as follows when all vertices that have a fan in value of greater or equal to 3: -

```
match("Fanin",[10, 7, 8, 9]).
match("Fanin",[42, 41, 43, 44, 45]).
```

13.3.1.2 Fan Out Models

The Variable Model Detection System found that vertices 1,10,20,24,28,32 and 36 are fan out vertices. The output from the Fan In model search part of the Variable Model Detection System is as follows when all vertices that have a fan out value of greater or equal to 3: -

```
match("Fanout",[1, 18, 2, 37, 41, 6]).
match("Fanout",[10, 11, 12, 13]).
```

```
match("Fanout",[20, 21, 22, 23]).
match("Fanout",[24, 25, 26, 27]).
match("Fanout",[28, 25, 26, 27, 29, 30, 31]).
match("Fanout",[32, 33, 34, 35]).
match("Fanout",[36, 33, 34, 35]).
```

As it can see vertex 10 is both a fan in and fan out vertex, it is therefore a candidate for a split 3 vertex.

13.3.1.3 Split 1 Models

A Split 1 model consists of two fan out vertices that fan out to common vertices. Hence vertices 32,36,24 and 28 are listed as being detected as fan out vertices. Split 1 models are used in the Split 3 model, hence later on in this thesis it is shown that vertices 24 and 28 are in the Split 3 model also. These were processed by the Variable Model Detection System to find the common vertices and the output from the system was: -

```
match("Split1",[32, 33, 34, 35, 36]).
match("Split1",[24, 25, 26, 27, 28]).
```

13.3.1.4 Split 2 Models

Due to the fact that split 2 models consist of a split1 model and a Fan Out model a split 2 model should consist of vertices already listed above. This is true of vertex 24. Therefore the output from the Variable Model Detection System is as follows: -

```
match("Split2",[24, 25, 26, 27, 28, 29, 30, 31]).
```

13.3.1.5 Split 3 Models

As mentioned above vertex 10 was a possible for this model. It has been detected using the variable mode detection system.

```
match("Split3",[10, 7, 8, 9, 11, 12, 13]).
```

13.3.1.6 Chain

Chains prove difficult to detect. It may be that they must only have a fan in value of 1 that means that a starting point is difficult to detect. It can also be seen below that a box has a chain present, hence the detection of the chain between vertices 14,15 and 16. Due to this chain should be given a low priority in any order they appear in a file. The Variable Model Detection System detected the following chains present in graph G: -

```
match("Chain",[14, 15, 21]).
match("Chain",[14, 15, 16]).
match("Chain",[2, 3, 4, 5]).
```

13.3.1.7 Chain to Fan Out Models

Chain to Fan Out models are difficult to detect because of reasons already discussed in the chain section above. However they proved a little more reliable than chains because they do not share any common features with the box model. The fact that the last vertex can fan out to more than one vertex means that it is more common. The Variable Model Detection System detected the above chain to Fan Out models present in graph G: -

```
match("ChainFanOut",[18, 19, 20, 21, 22, 23]).
match("ChainFanOut",[22, 24, 25, 26, 27]).
match("ChainFanOut",[23, 32, 33, 34, 35]).
match("ChainFanOut",[37, 38, 39, 40]).
match("ChainFanOut",[6, 7, 10, 11, 12, 13]).
```

13.3.1.8 Fixed Models – Box and Triangle

When running graph G through the Fixed Model Detection System the following output was produced: -

```
match("Box",[17,16,15,14]).
match("Triangle",[38,40,39]).
```

13.4 Match Analyser

Figure 93 provides an example of a representation file that is produced by the Match Analyser. It is one that is produced by inputting the matches into the Match Analyser in the order given in Chapter 6.

```
graph{
    node(9,0,0,0,0,"9","_","_","Black").
    node(8,0,0,0,0,"8","_","_","Black").
    node(7,0,0,0,0,"7","_","_","Black").
    node(6,0,0,0,0,"6","_","_","Black").
    node(5,0,0,0,0,"5","_","_","Black").
    node(45,0,0,0,0,"45","_","_","Black").
    node(44,0,0,0,0,"44","_","_","Black").
    node(43,0,0,0,0,"43","_","_","Black").
    node(42,0,0,0,0,"42","_","_","Black").
    node(41,0,0,0,0,"41","_","_","Black").
    node(40,0,0,0,0,"40","_","_","Black").
    node(4,0,0,0,0,"4","_","_","Black").
    node(39,0,0,0,0,"39","_","_","Black").
    node(38,0,0,0,0,"38","_","_","Black").
    node(37,0,0,0,0,"37","_","_","Black").
    node(36,0,0,0,0,"36","_","_","Black").
    node(35,0,0,0,0,"35","_","_","Black").
    node(34,0,0,0,0,"34","_","_","Black").
    node(33,0,0,0,0,"33","_","_","Black").
    node(32,0,0,0,0,"32","_","_","Black").
    node(31,0,0,0,0,"31","_","_","Black").
    node(30,0,0,0,0,"30","_","_","Black").
    node(3,0,0,0,0,"3","_","_","Black").
    node(29,0,0,0,0,"29","_","_","Black").
```

```
node(28,0,0,0,0,"28","_","_","Black").
node(27,0,0,0,0,"27","_","_","Black").
node(26,0,0,0,0,"26","_","_","Black").
node(25,0,0,0,0,"25","_","_","Black").
node(24,0,0,0,0,"24","_","_","Black").
node(23,0,0,0,0,"23","_","_","Black").
node(22,0,0,0,0,"22","_","_","Black").
node(21,0,0,0,0,"21","_","_","Black").
node(20,0,0,0,0,"20","_","_","Black").
node(2,0,0,0,0,"2","_","_","Black").
node(19,0,0,0,0,"19","_","_","Black").
node(18,0,0,0,0,"18","_","_","Black").
node(17,0,0,0,0,"17","_","_","Black").
node(16,0,0,0,0,"16","_","_","Black").
node(15,0,0,0,0,"15","_","_","Black").
node(14,0,0,0,0,"14","_","_","Black").
node(13,0,0,0,0,"13","_","_","Black").
node(12,0,0,0,0,"12","_","_","Black").
node(11,0,0,0,0,"11","_","_","Black").
node(10,0,0,0,0,"10","_","_","Black").
node(1,0,0,0,0,"1","_","_","Black").
edge(9,10,"_","directed","LineSolid","Black").
edge(8,10,"_","directed","LineSolid","Black").
edge(7,10,"_","directed","LineSolid","Black").
edge(6,7,"_","directed","LineSolid","Black").
edge(45,42,"_","directed","LineSolid","Black").
edge(44,42,"_","directed","LineSolid","Black").
edge(43,42,"_","directed","LineSolid","Black").
edge(41,42,"_","directed","LineSolid","Black").
edge(40,39,"_","directed","LineSolid","Black").
edge(4,5,"_","directed","LineSolid","Black").
edge(38,40,"_","directed","LineSolid","Black").
edge(38,39,"_","directed","LineSolid","Black").
edge(37,38,"_","directed","LineSolid","Black").
edge(36,35,"_","directed","LineSolid","Black").
edge(36,34,"_","directed","LineSolid","Black").
edge(36,33,"_","directed","LineSolid","Black").
edge(32,35,"_","directed","LineSolid","Black").
edge(32,34,"_","directed","LineSolid","Black").
edge(32,33,"_","directed","LineSolid","Black").
edge(3,4,"_","directed","LineSolid","Black").
edge(28,31,"_","directed","LineSolid","Black").
edge(28,30,"_","directed","LineSolid","Black").
edge(28,29,"_","directed","LineSolid","Black").
edge(28,27,"_","directed","LineSolid","Black").
edge(28,26,"_","directed","LineSolid","Black").
edge(28,25,"_","directed","LineSolid","Black").
edge(24,27,"_","directed","LineSolid","Black").
edge(24,26,"_","directed","LineSolid","Black").
edge(24,25,"_","directed","LineSolid","Black").
```

```
        edge(23,32,"_","directed","LineSolid","Black").
        edge(22,24,"_","directed","LineSolid","Black").
        edge(21,14,"_","directed","LineSolid","Black").
        edge(20,23,"_","directed","LineSolid","Black").
        edge(20,22,"_","directed","LineSolid","Black").
        edge(20,21,"_","directed","LineSolid","Black").
        edge(2,3,"_","directed","LineSolid","Black").
        edge(19,20,"_","directed","LineSolid","Black").
        edge(18,19,"_","directed","LineSolid","Black").
        edge(17,15,"_","directed","LineSolid","Black").
        edge(17,16,"_","directed","LineSolid","Black").
        edge(16,14,"_","directed","LineSolid","Black").
        edge(14,15,"_","directed","LineSolid","Black").
        edge(10,13,"_","directed","LineSolid","Black").
        edge(10,12,"_","directed","LineSolid","Black").
        edge(10,11,"_","directed","LineSolid","Black").
        edge(1,6,"_","directed","LineSolid","Black").
        edge(1,41,"_","directed","LineSolid","Black").
        edge(1,37,"_","directed","LineSolid","Black").
        edge(1,2,"_","directed","LineSolid","Black").
        edge(1,18,"_","directed","LineSolid","Black").
        algorithm("GT").
    }
    structure{
        name("Split2").
        algorithm("Split2").
        nodeused(24).
        nodeused(25).
        nodeused(26).
        nodeused(27).
        nodeused(28).
        nodeused(29).
        nodeused(30).
        nodeused(31).
    }
    structure{
        name("Split1").
        algorithm("Split1").
        nodeused(32).
        nodeused(33).
        nodeused(34).
        nodeused(35).
        nodeused(36).
    }
    structure{
        name("Split3").
        algorithm("Split3").
        nodeused(10).
        nodeused(7).
        nodeused(8).
```

```
        nodeused(9).
        nodeused(11).
        nodeused(12).
        nodeused(13).
    }
    structure{
        name("ChainFanOut").
        algorithm("GT").
        nodeused(18).
        nodeused(19).
        nodeused(20).
        nodeused(21).
        nodeused(22).
        nodeused(23).
    }
    structure{
        name("ChainFanOut").
        algorithm("GT").
        nodeused(37).
        nodeused(38).
        nodeused(39).
        nodeused(40).
    }
    structure{
        name("Box").
        algorithm("Box").
        nodeused(17).
        nodeused(16).
        nodeused(15).
        nodeused(14).
    }
    structure{
        name("Fanin").
        algorithm("FanIn").
        nodeused(42).
        nodeused(41).
        nodeused(43).
        nodeused(44).
        nodeused(45).
    }
    structure{
        name("Chain").
        algorithm("GT").
        nodeused(2).
        nodeused(3).
        nodeused(4).
        nodeused(5).
    }
```

Figure 93 - The graph representation file

13.5 GIN Output File

The display system for the ANHOF system is Graph Tool, this interprets GIN files. Therefore the output from the ANHOF system should be a GIN file. In Figure 94 the GIN output file for the Graph G is shown.

(object) 1 0 241 54 262 (1) (_) (_) object
(object) 2 114 457 168 478 (2) (_) (_) object
(object) 3 235 457 289 478 (3) (_) (_) object
(object) 4 347 458 401 479 (4) (_) (_) object
(object) 5 472 459 526 480 (5) (_) (_) object
(object) 6 113 27 167 48 (6) (_) (_) object
(object) 7 235 28 289 49 (7) (_) (_) object
(object) 8 235 89 289 110 (8) (_) (_) object
(object) 9 235 58 289 79 (9) (_) (_) object
(object) 10 345 58 406 79 (10) (_) (_) object
(object) 11 463 89 524 110 (11) (_) (_) object
(object) 12 463 58 524 79 (12) (_) (_) object
(object) 13 463 28 524 49 (13) (_) (_) object
(object) 14 587 406 648 427 (14) (_) (_) object
(object) 15 587 459 648 480 (15) (_) (_) object
(object) 16 710 406 771 427 (16) (_) (_) object
(object) 17 710 459 771 480 (17) (_) (_) object
(object) 18 110 331 171 352 (18) (_) (_) object
(object) 19 231 332 292 353 (19) (_) (_) object
(object) 20 346 333 407 354 (20) (_) (_) object
(object) 21 469 406 530 427 (21) (_) (_) object
(object) 22 469 334 530 355 (22) (_) (_) object
(object) 23 469 209 530 230 (23) (_) (_) object
(object) 24 587 334 648 355 (24) (_) (_) object
(object) 25 710 361 771 382 (25) (_) (_) object
(object) 26 710 334 771 355 (26) (_) (_) object
(object) 27 710 307 771 328 (27) (_) (_) object
(object) 28 830 334 891 355 (28) (_) (_) object
(object) 29 937 362 998 383 (29) (_) (_) object
(object) 30 937 334 998 355 (30) (_) (_) object
(object) 31 937 307 998 328 (31) (_) (_) object
(object) 32 586 210 647 231 (32) (_) (_) object
(object) 33 710 238 771 259 (33) (_) (_) object
(object) 34 710 210 771 231 (34) (_) (_) object
(object) 35 710 179 771 200 (35) (_) (_) object
(object) 36 829 210 890 231 (36) (_) (_) object
(object) 37 111 241 172 262 (37) (_) (_) object
(object) 38 232 242 293 263 (38) (_) (_) object
(object) 39 344 286 405 307 (39) (_) (_) object
(object) 40 344 188 405 209 (40) (_) (_) object

(object) 41 111 118 172 139 (41) (_) (_) object
(object) 42 232 158 293 179 (42) (_) (_) object
(object) 43 111 201 172 222 (43) (_) (_) object
(object) 44 111 172 172 193 (44) (_) (_) object
(object) 45 111 147 172 168 (45) (_) (_) object
(link) 1 18 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 1 2 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 1 37 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 1 41 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 1 6 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 10 11 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 10 12 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 10 13 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 14 15 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 16 14 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 17 15 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 17 16 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 18 19 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 19 20 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 2 3 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 20 21 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 20 22 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 20 23 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 21 14 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 22 24 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 23 32 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 24 25 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 24 26 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 24 27 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 28 25 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 28 26 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 28 27 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 28 29 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 28 30 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 28 31 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 3 4 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 32 33 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 32 34 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 32 35 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 36 33 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 36 34 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 36 35 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 37 38 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 38 39 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 38 40 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 4 5 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 40 39 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 41 42 0 0 0 0 0 0 (_) (directed) (LineSolid) link
(link) 43 42 0 0 0 0 0 0 (_) (directed) (LineSolid) link

```
( link ) 44 42 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 45 42 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 6 7 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 7 10 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 8 10 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
( link ) 9 10 0 0 0 0 0 0 ( _ ) ( directed ) ( LineSolid ) link
```

Figure 94 - The GIN output file from the ANHOF system.

14. Bibliography and References

[1] IEEE Standard Glossary of Software Engineering Terminology: IEEE, 1983.

[2] "Graphic File Formats FAQ", 12th October 2000, Available From http://www.cs.ruu.nl/wais/html/na-dir/graphics/fileformats-faq/{part1.html, part2.html, part3.html, part4.html}.

[3] G. M. Adelson-Velskii and E. M. Landis, "An Algorithm for the Organization of Information," Soviet Mathematics Dokl, vol. 3, pp. 1259-1263, 1962.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, Data Structures and Algorithms, 2nd Edition ed: Addison-Wesley, 1987.

[5] ANSI, Standard Flowchart Symbols and their use in Information Processing (X3.5). New York: American National Standards Institute, 1971.

[6] L. Babai, "Moderately Exponential Bound for Graph Isomorphism," in Fundamentals of Computation Theory, vol. 117, Lecture Notes in Computer Science, F. Gecseg, Ed. Berlin: Springer-Verlag, 1981, pp. 34-50.

[7] N. S. Barghouti, J. M. Mocenigo, and W. Lee, "Grappa : A GRAPh PAckage in Java," in Graph Drawing, vol. 1353, Lecture Notes in Computer Science, G. DiBattista, Ed. Berlin: Springer-Verlag, 1997, pp. 336-349.

[8] C. Batini, L. Furlani, and E. Nardelli, "What is a Good Diagram? A Pragmatic Approach," presented at 4th International Conference on Entity Relationship Approach, Chicago, 1985.

[9] J. Bentley, "Column 9: LITTLE LANGUAGES," in More Programming Perls:Confessions of a Coder: Addison- Wesley, 1988, pp. 83-100.

[10] J. Berry, N. Dean, M. Golderg, G. Shannon, and S. Skiena, "Graph Drawing and Manipulation with LINK," in Graph Drawing, vol. 1353, Lecture Notes in Computer Science, G. DiBattista, Ed. Rome: Springer-Verlag, 1997, pp. 425-437.

[11] F. Bertault, "ADOCS: a Drawing System for Generic Combinatorial Structures," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 24-27.

[12] P. Bertolazzi, G. DiBattista, and G. Liotta, "Parametric Graph Drawing," IEEE Transactions on Software Engineering, vol. 21(8), pp. 662-673, 1995.

[13] S. N. Bhatt and F. T. Leighton, "A Framework for Solving VLSI Graph Layout Problems," Journal of Computer and System Sciences, vol. 28, pp. 300-343, 1984.

[14] A. Bloesch, "Aesthetic Layout of Generalized Trees," Software-Practice and Experience, vol. 23(8), pp. 817-827, 1993.

[15] J. Blythe, C. McGrapth, and D. Krackhardt, "The Effect of Graph Layout on Inference form Social Network Data," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 40-51.

[16] T. Bodhuin, "An Interaction Paradigm For Impact Analysis", M.S.c Thesis, University of Durham, Durham, 1995.

[17] B. W. Boehm, "A Spiral Model of Software Development and Enhancement," IEEE Computer, vol. 21(5), pp. 61-72, 1988.

[18] C. Bohm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules," Communications of the ACM, vol. 9(5), pp. 366-371, 1966.

[19] C. Boldyreff, E. L. Burd, R. M. Hather, R. E. Mortimer, M. Munro, and E. J. Younger, "The AMES Approach to Application Understanding: A Case Study," in The Proceedings of the International Conference on Software Maintenance 1995: IEEE, 1994, pp. 182-191.

[20] K. S. Booth and G. S. Lueker, "Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms.," Journal of Computer System Science, vol. 13, pp. 335-379, 1976.

[21] F. J. Brandenberg, "Designing Graph Drawings by Layout Graph Grammars," in Graph Drawing, vol. 894, Lecture Notes in Computer Science, R. Tamassia and I. G. Tollis, Eds. Passau: Springer-Verlag, 1995, pp. 416-427.

[22] F. J. Brandenburg, "Layout Graph Grammars: The Placement Approach," , vol. 532, Lecture Notes in Computer Science: Springer-Verlag, 1991, pp. 144-156.

[23] F. J. Brandenburg, M. Himsolt, and C. Rohrer, "An Experimental Comparison of Force Directed and Randomized Graph Drawing Algorithms," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 76-87.

[24] S. S. Bridgeman, J. Fanto, A. Garg, R. Tamassia, and L. Vismara, "InteractiveGiotto : An Algorithm for Interactive Orthogonal Graph Drawing," in Graph Drawing, vol. 1353, Lecture Notes in Computer Science, G. DiBattista, Ed. Rome: Springer-Verlag, 1997, pp. 303-308.

[25] R. Brooks, "Towards a Theory of the Comprehension of Computer-Programs," International Journal of Man-Machine Studies, vol. 18(6), pp. 543-554, 1983.

[26] H. Bunke and B. T. Messmer, "Recent Advances in Graph Matching," International Journal of Pattern Recognition and Artificial Intelligence, vol. 11(1), pp. 169-203, 1997.

[27] E. L. Burd, P. S. Chan, I. M. M. Duncan, M. Munro, and P. Young, "Improving Visual Representations of Code," University of Durham, Durham, Technical Report 10/96, 1996.

[28] L. Buti, G. DiBattista, G. Liotta, E. Tassinari, F. Vargiu, and L. Vismara, "GD-Workbench: A System for Prototyping and Testing Graph Drawing Algorithms," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 111-122.

[29] Y. Carbonneaux, J. Laborde, and R. M. Madani, "CABRI-Graph: A Tool for Research and Teaching in Graph Theory," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 123-126.

[30] M. Carpano, "Automatic Display of Hierarchized Graphs for Computer - Aided Decision Analysis," IEEE Transactions Of Systems Man and Cybernetics, vol. 10(11), pp. 705-715, 1980.

[31] N. Chapin, Flowcharts. Princeton: Auerberg, 1971.

[32] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa, "A Linear Algorithm for Embedding Planar Graphs Using PQ Trees," Journal of Computer and System Sciences, vol. 30, pp. 54-76, 1985.

[33] E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, vol. 7(1), pp. 13-17, 1990.

[34] W. F. Clocksin and C. S. Mellish, Programming in Prolog, 4th Edition ed. Berlin: Springer-Verlag, 1998.

[35] M. K. Coleman and D. Stott-Parker, "Aesthetics-Based Graph Layout for Human Consumption," Software Practice and Experience, vol. 26(12), pp. 1415-1438, 1995.

[36] D. G. Corneil and C. C. Gotlieb, "An Efficient Algorithm for Graph Isomorphism," Journal of the ACM, vol. 17(1), pp. 51-64, 1979.

[37] B. Cornelius, Programming with TopSpeed Modula-2. Wokingham: Addison-Wesley, 1991.

[38] N. Cunnliff and R. P. Taylor, "Graphical vs. Textual Representation: An Empirical Study of Novices' Program Comprehension," in Empirical Studies of Programmers: Second Workshop, Human computer Interaction, G. M. Olson, S.Sheppard, and E.Soloway, Eds., 1987, pp. 114-131.

[39] M. Dao, M. Harib, J. P. Richard, and D. Tallot, "CABRI : An Interactive System For Graph Manipulation.," in Graph-Theoretic Concepts in Computer Science, vol. 246, Lecture Notes in Computer Science, G. Tinhofer and G. Schmidt, Eds.: Springer-Verlag -Verlag, 1986, pp. 58-67.

[40] R. Davidson and D. Harel, "Drawing Graphs Nicely Using Simulated Annealing," ACM Transactions on Graphics, vol. 15(4), pp. 301-331, 1996.

[41] E. Dengler and W. Cowan, "Human Perception of Laid-out Graphs," in Graph Drawing, vol. 1547, Lecture Notes in Computer Science: Springer-Verlag, 1998, pp. 441-443.

[42] G. DiBattista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for Drawing Graphs: an Annotated Bibliography," Computational Geometry: Theory and Applications, vol. 4, pp. 235-282, 1994. Available From http://www.cs.brown.edu/people/rt/gd-biblio.html.

[43] G. DiBattista, P. Eades, R. Tamassia, and I. G. Tollis, Graph Drawing - Algorithms for theVisualization of Graphs: Prentice Hall, 1999.

[44] G. DiBattista, A. Garg, G. Liotta, A. Parise, R. Tamassia, E. Tassinari, F. Vargiu, and L. Vismara, "Drawing Directed Acyclic Graphs: An Experimental Study," in Graph Drawing, vol. 1190, Lecture Notes In Computer Science, S. North, Ed.: Springer-Verlag, 1996, pp. 76-91.

[45] G. DiBattista, G. Liotta, A. Garg, R. Tamassia, E. Tassinari, and F. Vargiu, "An Experimental Comparison of Four Drawing Algorithms," Computational Geometry : Theory and Applications, vol. 7, pp. 202-325, 1995.

[46] C. Ding and P. Mateti, "A Framework for the Automated Drawing of Data Structure Diagrams," IEEE Transactions on Software Engineering, vol. 16(5), pp. 543-557, 1990.

[47] D. P. Dobkin, E. R. Gansner, E. Kotsofios, and S. C. North, "Implementing a General-Purpose Edge Router," in Graph Drawing, vol. 1353, Lecture Notes in Computer Science, G. DiBattista, Ed. Rome: Springer-Verlag, 1997, pp. 262-271.

[48] D. Dodson, "COMAIDE: Information Visualization using Cooperative 3D Diagram Layout," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 190-201.

[49] U. Dogrusoz and B. Madden, "Circular Layout in the Graph Layout Toolkit," in Graph Drawing, vol. 1190, Lecture Notes in Computer Science, S. North, Ed.: Springer-Verlag, 1996, pp. 76-91.

[50] P. Eades, "A Heuristic for Graph Drawing," Congressus Numeratium, vol. 42, pp. 149-160, 1984.

[51] M. Elson, Concepts of Programming Languages. Chicago: Science Research Associates, 1973.

[52] V. Engelson, "Call Graph Drawing Interface", 18th June 2000 Available From http://www.ida.liu.se/~vaden/cgdi/.

[53] B. Everitt, Cluster Analysis, 1st ed. London: Heinemann, 1974.

[54] B. Everitt, Cluster Analysis, 3rd ed. London: Arnold, 1993.

[55] R. E. Fairley, Software Engineering Concepts. New York: McGraw-Hill, 1985.

[56] N. Fenton and G. Hill, Systems Construction and Analysis : A Mathematical and Logical Framework. London: McGraw-Hill, 1993.

[57] N. E. Fenton, Software Metrics: A Rigorous Approach, 1st ed: Chapman and Hall, 1991.

[58] N. E. Fenton and S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, 2nd ed: PWS Publishing International Thomson Computer Press, 1996.

[59] N. E. Fenton and R. W. Whitty, "Axiomatic Approach to Software Metrication through Program Decomposition," The Computer Journal, vol. 29(4), pp. 330-339, 1986.

[60] R. K. Fjeldstad and W. T. Hamlen, "Application Program Maintenance Study - A Report to our Respondent," in Tutorial On Software Maintenance, G. Parikh and N. Zvegintzov, Eds.: IEEE, 1983, pp. 13-25.

[61] H. W. Fowler, F. G. Fowler, and J. Pearsall, "The Concise Oxford Dictionary," . Oxford: Oxford University Press, 1999.

[62] T. M. J. Frauchterman and E. M. Reingold, "Graph Drawing by Force Directed Placement," Software Practice and Experience, vol. 21(11), pp. 1129-1164, 1991.

[63] A. Frick, A. Ludwig, and H. Mehldau, "A Fast Adaptive Layout Algorithm for Undirected Graphs," in Graph Drawing, vol. 894, Lecture Notes in Computer Science, R. Tamassia and I. G. Tollis, Eds.: Springer-Verlag, 1994, pp. 388-403.

[64] M. Frohlich and M. Werner, "Demonstration of the Interactive Graph Visualization System daVinci," in Graph Drawing, vol. 894, Lecture Notes in Computer Science, R. Tamassia and I. G. Tollis, Eds.: Springer-Verlag, 1994, pp. 266-269.

[65] M. Frohlich and M. Werner, daVinci V1.4 Transistion Guide, 1996.

[66] C. P. Gane, Structured Systems Analysis: Tools and Techniques: Prentice-Hall, 1979.

[67] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo, "A Technique for Drawing Directed Graphs," IEEE transactions on Software Engineering, vol. 19(3), pp. 214 - 230, 1993.

[68] E. R. Gansner, S. C. North, and K. P. Vo, "DAG - A Program that Draws Directed Graphs," Software - Practice and Experience, vol. 18(11), pp. 1047-1062, 1988.

[69] M. R. Garey and D. S. Johnson, Computers and Intractability a Guide to the Theory of NP-Completeness. New York: W.H. Freeman, 1979.

[70] A. Garg and R. Tamassia, "GIOTTO3D : A system for Visualizing Hierarchical Structures in 3D," in Graph Drawing, vol. 1190, Lecture Notes In Computer Science, S. North, Ed.: Springer-Verlag, 1996, pp. 193-200.

[71] G. Gati, "Further Annotated Bibliography on the Isomorphism Disease," Journal of Graph Theory, vol. 3, pp. 95-109, 1979.

[72] H. H. Goldstine and J. von Newmann, Planning and Coding Problems for an Electronic Computing Instrument, vol. 1-3. Princeton, N.J.: D Van Nostrand Co, 1946.

[73] J. Gross and J. Yellen, Graph Theory and Its Applications: CRC Press, 1999.

[74] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call Graph Construction In Object-Oriented Languages," ACM Sigplan Notices, vol. 32(10), pp. 108-124, 1997.

[75] R. N. Haber, "How We Remember What We See," Scientific American, vol. 105, 1970.

[76] S. Henry and D. Kafura, "Software Structure Metrics Based On Information-Flow," IEEE Transactions on Software Engineering, vol. 7(5), pp. 510-518, 1981.

[77] M. Himsolt, "GraphEd: A Graphical Platform for the Implementation of Graph Algorithms," in Graph Drawing, vol. 894, Lecture Notes in Computer Science, R. Tamassia and I. G. Tollis, Eds.: Springer-Verlag, 1994, pp. 182-193.

[78] M. Himsolt, "GML: A Portable Graph File Format", 02/12/1998 Available From http://www.fmi.uni-passau.de/Graphlet/GML/gml-tr.html.

[79] M. Himsolt, "GML-Graph Modelling Language", 1st December 1998 Available From http://www.fmi.uni-passau.de/archive/archive.theory/ftp/graphlet/GML.ps.gz.

[80] M. Himsolt, "The Graphlet System (System Demonstration)," in Graph Drawing, vol. 1190, Lecture Notes In Computer Science, S. North, Ed.: Springer-Verlag, 1996, pp. 233-240.

[81] C. Hoffmann, Group-theoretic Algorithms and Graph Isomorphism, vol. 136: Springer-Verlag, 1982.

[82] C. Hsu, "Minimum - Via Topological Routing," IEEE Transactions on Computer-Aided Design, vol. 2(4), pp. 235-246, 1983.

[83] J. Ignatowicz, "Drawing Force Directed Graphs Using Optigraph," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 333-336.

[84] A. K. Jain and R. C. Dubes, Algorithms for Clustering Data. Englewood Cliffs, New Jersey: Prentice Hall, 1988.

[85] B. A. Jeffries, "Comparison of Debugging Behaviour of novice and expert programmers.," Department of Psychology, Carnegie - Mellon Universirty., Pittsburgh, P.A. 1982.

[86] R. Johnson, D. Pearson, and K. Pingali, "The Program Structure Tree - Computing Control Regions in Linear-Time," Sigplan Notices, vol. 29(6), pp. 171-185, 1994.

[87] M. Junger, P. Mutzel, and S. Nafer, "AGD - Algorithms for Graph Drawing User Manual Version 1.0.1", 1999, Available From http://www.mpi-sb.mpg.de/AGD/.

[88] Y. Kahn and T. Baylis, "A Survey of Graph Drawing Algorithms Based on Graph Grammars," in Graph Grammars and Their Application to Computer Science, vol. 1073, Lecture Notes in Computer Science, J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, Eds. Williamsburg, VA, USA: Springer-Verlag, 1994.

[89] T. Kamada and S. Kawai, "An Algorithm for Drawing General Undirected Graphs," Information Processing Letters, vol. 31, pp. 7-15, 1989.

[90] S. Kamin and D. Hyatt, "A Special-Purpose Language for Picture-Drawing," presented at Conference On Domain Specific Language, Santa Barbara, California, 1997.

[91] M. Kaul, "Specification Of Error Distances for Graphs by Precedence Graph Grammars and Fast Recognition of Similarity," in Graph-Theoretic Concepts in Computer Science, vol. 246, Lecture Notes in Computer Science, G. Tinhofer and G. Schmidt, Eds.: Springer-Verlag, 1986, pp. 29-40.

[92] B. W. Kernighan, "PIC - A Language for Typesetting Graphics," Software - Practice and Experience, vol. 12, pp. 1-21, 1982.

[93] B. W. Kernighan and D. M. Ritchie, The C Programming Language., 2nd ed, 1989.

[94] D. Kinloch and M. Munro, "A Combined Representation for the Maintenance of C Programs," in Proceedings of IEEE 2nd Workshop on Program Comprehension. Capri Italy: IEEE, 1993, pp. 118-127.

[95] D. A. Kinloch, "A Combined Representation for the Maintenance of C Programs", Ph.d. Thesis, University of Durham, Durham, 1995.

[96] B. A. Kitchenham, "Evaluating Software Engineering Methods and Tool Part 1: The Evaluation Context and Evaluation Methods," ACM SIGSOFT, vol. 21(1), pp. 11-15, 1996.

[97] C. R. Knight, "Visualisation for Program Comprehension: Information and Issues," Department of Computer Science, University of Durham, Durham, Technical Report 12/98, 1998.

[98] J. Kobler, U. Schoning, and J. Toran, The Graph Isomorphism Problem : Its Structural Complexity. Boston: Birkhauser, 1993.

[99] T. D. Korson and V. K. Vishnavi, "An Empirical Study of the Effects of Modularity on Program Modifiability," in Empirical Studies of Programmers, Human/Computer Interaction, E. Soloway and S. Iyengar, Eds. Washington: Ablex, 1986, pp. 168-186.

[100] C. Kosak, J. Marks, and S. Shieber, "Automating the Layout of Network Diagrams With Specified Visual Organization," IEEE Transactions on Systems, Man and Cybernetics, vol. 24(3), pp. 440-455, 1994.

[101] H. Ledgard and M. Marcotty, "A Genealogy Of Control Structures," Communications of the ACM, vol. 18(11), pp. 629-639, 1975.

[102] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and Laws of Software Evolution - The Nineties View," presented at Elements of Software Process Assessment and Improvement, Albuquerque, New Mexico, 1997.

[103] I. Lemke and G. Sander, "Visualisation of Compiler Graphs : Design Report and Documentation," Universitat des Saarlandes, Technical 30/04/1998 1994.

[104] U. Lichtblau, "Flusgraphgammatiken," AusDam Fachbereivh Informatik,University of Oldenburg, Germany, Technical Report 3/90, March 1990.

[105] B. P. Lientz and E. B. Swanson, Software Maintenance Management. Reading MA: Addison Wesley, 1980.

[106] T. Lin and P. Eades, "Integration of Declarative and Algorithmic Approaches for Layout Creation," in Lecture Notes in Computer Science, vol. 894, R. Tamassia and I. G. Tollis, Eds.: Springer-Verlag, 1994, pp. 376-387.

[107] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Metal Models and Software Maintenance," in Empirical Studies of Programmers,

Human/Computer Interaction, E. Soloway and S. Iyengar, Eds. Ablex Norwood, N.J., 1986, pp. 80-98.

[108] B. Madden, P. Madden, S. Powers, and M. Himsolt, "Portable Graph Layout and Editing," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 385-395.

[109] T. J. McCabe, "A Complexity Measure," IEEE Transaction of Software Engineering, vol. 2(4), pp. 308-322, 1976.

[110] C. McCreary, F. Shieh, and H. Gill, "CG: A Graph Drawing System Using Graph Grammar Parsing," in Graph Drawing, vol. 894, Lecture Notes in Computer Science, R. Tamassia and I. G. Tollis, Eds.: Springer-Verlag, 1994, pp. 270-273.

[111] K. Mehlhorn and N. Stefan, LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge: Cambridge University Press, 1999.

[112] E. B. Messinger, "Automatic Layout of Large Directed Graphs", Ph.D. Thesis, University of Washington, 1989.

[113] E. N. Messinger, L. A. Rowe, and R. R. Henry, "A Divide-and-Conquer for the Automatic Layout of Large Directed Graphs," IEEE Transactions on Systems, Man and Cybernetics, vol. 21(1), pp. 1-11, 1991.

[114] B. T. Messmer, "Efficient Graph Matching Algorithms for Preprocessed Model Graphs", PhD Thesis, Univerisity of Bern, Switzerland, 1995. Available From http://iamwww.unibe.ch/~fkiwww/publications/index.html.

[115] B. T. Messmer and H. Bunke, "A Decision Tree Approach to Graph and Subgraph Isomorphism Detection," Pattern Recognition, vol. 32(12), pp. 1979-1998, 1999.

[116] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Schneiderman, "Program Indentation and Comprehensibility," Communications of the ACM, vol. 26(11), pp. 861-867, 1983.

[117] M. Munro, E. L. Burd, P. Chan, and P. Young, "The Shape of Software to Come : Results of Preliminary Investigations of the VRG," in Proceedings of 2nd UK workshop on Program Comprehension. Durham: University of Durham, 1996.

[118] G. C. Murphy, D. Notkin, and E. S.-C. Lan, "An Empirical Study of Static Call Graph Extractors," in Proceedings of the 18th International Conference on Software Engineering. Berlin Germany: IEEE, 1996, pp. 90-99.

[119] P. Mutzel, C. Gutwenger, R. Brockenauer, S. Fialko, G. Klau, M. Kruger, T. Ziegler, S. Naher, D. Alberts, D. Ambras, G. Koch, M. Junger, C. Buchheim, and S. Leipert, "A library of algorithms for graph drawing," Lecture Notes in Computer Science, vol. 1547, pp. 456-457, 1998. Available From http://www.mpi-sb.mpg.de/AGD/.

[120] B. A. Myers, "Taxonomies of Visual Programming and Program Visualization," Journal of Visual Languages and Computing., vol. 1, pp. 97-123, 1990.

[121] P. Naur and B. Randell, "Software Engineering:A Report on a Conference Sponsored by the NATO Science Committee," : NATO, 1969.

[122] W. M. Newham, "A Prototype Low Cost Single User Graphic System," in Graphic Languages, F. Nake and A. Rosenfield, Eds. Amsterdam: North-Holland Publishing Company, 1972, pp. 291-301.

[123] S. C. North, "Incremental Layout in DynaDAG," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 409-418.

[124] M. R. Paige, "On Partitioning Program Graphs," IEEE Transactions on Software Engineering, vol. SE-3(6), pp. 386-393, 1977.

[125] A. Papakostas, J. M. Six, and I. G. Tollis, "Experimental and Theoretical Results in Interactive Orthogonal Graph Drawing," in Graph Drawing, vol. 1190, Lecture Notes In Computer Science, S. North, Ed.: Springer-Verlag, 1996, pp. 371-386.

[126] A. Papakostas and I. G. Tollis, "Issues In Interactive Orthogonal Graph Drawing," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 419-430.

[127] A. Papakostas and I. G. Tollis, "Interactive Orthogonal Graph Drawing," IEEE Transactions on Computers, vol. 47(11), pp. 1297-1309, 1998.

[128] M. Patrignani and F. Vargiu, "3DCube : A Tool for Three Dimensional Graph Drawing," in Graph Drawing, vol. 1353, Lecture Notes in Computer Science, G. DiBattista, Ed. Rome: Springer-Verlag, 1997, pp. 284-290.

[129] F. N. Paulisch and W. F. Tichy, "EDGE: An Extendible Graph Editor," Software - Practice and Experience, vol. 20(S1), pp. 63-88, 1990.

[130] S. L. Pfleeger, "Design and Analysis in Software Engineering Part 1: The Language of Case Studies and Formal Experiments," ACM SIGSOFT, vol. 19(4), pp. 16-20, 1994.

[131] A. D. Polimeni and H. J. Straight, Foundations of Discrete Mathematics. Belmont: Brooks/Cole, 1985.

[132] T. W. Pratt and M. V. Zelkowitz, Programming Languages- Design and Implementation. New Jersey: Prentice Hall, 1996.

[133] B. A. Price, R. M. Baecker, and I. S. Small, "A Principle Taxonomy of Software Visualization," Journal of Visual Languages and Computing, vol. 4(3), pp. 211-266, 1992.

[134] L. B. Protsko, P. G. Sorenson, and J. P. Tremblay, "Mondrian - System For Automatic-Generation of Dataflow Diagrams," Information and Software Technology, vol. 31(9), pp. 456-471, 1989.

[135] M. H. Protter and C. B. Morray, Calculus with Analytic Geometry : A First Course: Addison Wesley, 1977.

[136] H. Purchase, "Which Aesthetic Has the Greatest Effect on Human Understanding?," in Graph Drawing, vol. 1353, Lecture Notes in Computer Science, G. DiBattista, Ed. Rome: Springer-Verlag, 1997, pp. 248-251.

[137] H. C. Purchase, "Performance of layout algorithms: Comprehension, not computation," Journal of Visual Languages and Computing, vol. 9(6), pp. 647-657, 1998.

[138] H. C. Purchase, R. F. Cohen, and M. James, "Validating Graph Drawing Aesthetics," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 435-446.

[139] H. C. Purchase and D. Leonard, "Graph Drawing Aesthetic Metrics," Department of Computer Science, University of Queensland, Technical Report 361, 1996.

[140] N. R. Quinn and M. A. Breuer, "A Forced Directed Component Placement procedure for Printed Circuit Boards," IEEE Transactions on Circuits and Systems, vol. 26(6), pp. 377-388, 1979.

[141] R. C. Read and D. G. Corneil, "The Graph Isomorphism Disease," Journal of Graph Theory, vol. 1, pp. 339-363, 1977.

[142] M. G. Rekoff, "On Reverse Engineering," IEEE Transactions of Systems Man and Cybernetics, vol. 15(2), pp. 244-252, 1985.

[143] D. J. Robson, K. H. Bennett, B. J. Cornelius, and M. Munro, "Approaches to Program Comprehension," Journal of Systems and Software, vol. 14(2), pp. 79-84, 1991.

[144] W. W. Royce, "Managing the Development of Large Software Systems," in Proceedings WESTCON. San Francisco, CA, 1970.

[145] R. G. Ryder, "Constructing a Call Graph of a Program," IEEE Transactions of Software Engineering, vol. SE-5(3), pp. 216-225, 1979.

[146] G. Sander, "Graph Layout through the VCG Tool," Universitat des Saarlandes, Technical Report A03/94, October 4,1994 1994.

[147] G. Sander, "Layout of Compound Directed Graphs," Universitat des Saarlandes, Saarbrucken, Technical Report A/03/96, June 5 1996.

[148] G. Sander, M. Alt, C. Ferdinand, and R. Willhelm, "CLAX - A Visualized Compiler," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 458-462.

[149] R. Sato, "Meaning of Dataflow Diagram and Entity Life History - A Systems Theoretic Foundation for Information Systems Analysis .," IEEE Transactions On Systems Man and Cybernetics Part a-Systems and Humans, vol. 27(1), pp. 11-22, 1997.

[150] B. Schneiderman, Software Psychology: Winthrop, 1980.

[151] B. Schneiderman, R. Meyer, D. McKay, and P. Heller, "Experimental Investigations of Utility of Detailed Flowcharts in Programming," Communications of the ACM, vol. 20(6), pp. 373-381, 1977.

[152] M. Skorsky, "TOSCANA Management System for Conceptual Data," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 483-486.

[153] D. D. Smith, Designing Maintainable Software: Springer-Verlag, 1999.

[154] I. Sommerville, Software Engineering, 5TH Ed. ed: Addison-Wesley, 1996.

[155] J. Soukup, "Circuit Layout," Proceedings of the IEEE, vol. 69(10), pp. 1281-1304, 1981.

[156] T. A. Standish, "An Essay on Software Reuse," IEEE Transactions in Software Engineering, vol. 10(5), pp. 494-497, 1984.

[157] K. Sugiyama and K. Misue, "A Simple and Unified Method for Drawing Graphs: Magnetic-Spring Algorithm," in Graph Drawing, vol. 894, Lecture Notes in Computer Science, R. Tamassia and I. G. Tollis, Eds.: Springer-Verlag, 1994, pp. 364-375.

[158] K. Sugiyama and K. Misue, "A Generic Compound Graph Visualizer/Manipulator : D- ABDUCTOR," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 500-503.

[159] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for Visual Understanding of Hierarchical System Structures," IEEE Transactions on Systems, Man, and Cybernetics, vol. 11(2), pp. 110-125, 1981.

[160] R. Tamassia, "On Embedding a Graph in the Grid with the Minimum Number of Bends," SIAM Journal of Computing, vol. 16(3), pp. 421-444, 1987.

[161] R. Tamassia, "Advances in the Theory and Practice of Graph Drawing," Theoretical Computer Science, vol. 217(2), pp. 235-254, 1999.

[162] R. Tamassia, G. DiBattista, and C. Batini, "Automatic Graph Drawing and Readability of Diagrams," IEEE Transactions on Systems,Man and Cybernetics, vol. 18(1), pp. 61-79, 1988.

[163] D. Tunkelang, "A Practical Approach to Drawing Undirected Graphs," School of Computer Science, Carnegie Mello University, Pittsburge, Technical Report CMU-CS-94-161, June 1994.

[164] J. R. Ullman, "An Algorithm for Subgraph Isomorphism," Journal of the ACM, vol. 23(1), pp. 31-42, 1976.

[165] J. G. Vaucher, "Pretty Printing Of Trees," Software Practice And Experience, vol. 10, pp. 554-561, 1980.

[166] J. Q. Walker, "A Node-positioning Algorith for General Tree," Software Practice and Experience, vol. 20(7), pp. 685-705, 1990.

[167] J. N. Warfield, "On Arranging Elements of a Hierarchy in Graphic Form," IEEE Transactions on Systems, Man and Cybernetics, vol. 3(2), pp. 121-132, 1973.

[168] J. N. Warfield, "Crossing Theory and Hierarchy Mapping," IEEE Transactions on Systems, Man and Cybernetics, vol. SMC -7(3), pp. 305-323, 1977.

[169] R. Webber and A. Scott, "GOVE Grammar-Oriented Visualisation Environment," in Graph Drawing, vol. 1027, Lecture Notes in Computer

Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 516-519.

[170] M. A. Weiss, Data Structures and Algorithm Analysis: Benjamin/Cummings, 1992.

[171] C. Wetherell and A. Shannon, "Tidy Drawings of Trees," IEEE Transactions on Software Engineering, vol. 5(5), pp. 514-520, 1979.

[172] S. Wiedenbeck, "Processes in Computer Program Comprehension," in Empirical Studies of Programmers, Human/Computer Interaction, E.Soloway and S.Iyengar, Eds. Washington: Ablex Publiching, 1986, pp. 48-57.

[173] J. Wielemaker, "SWI Prolog - Home", Available From http://www.swi.psy.uva.nl/projects/SWI-Prolog/.

[174] G. J. Wills, "NicheWorks - Interactive Visualization of Very Large Graphs," in Graph Drawing, vol. 1353, Lecture Notes in Computer Science, G. DiBattista, Ed. Rome: Springer-Verlag, 1997, pp. 403-414.

[175] D. Wilson, "Forms of Hierarchy: A Selected Bibliography," General Systems, vol. 14, pp. 3-15, 1969.

[176] R. J. Wilson, Introduction to Graph Theory, 4th ed: Longman, 1997.

[177] N. Wirth, Algorithms + Data Structures = Programs, 1st ed. Eagle Cliffs, New Jersey: Prentice-Hall, 1976.

[178] S. N. Woodfield, S. E. Dunsmore, and V. Y. Shren, "The Effect of Modularization and Comments on Program Comprehension," in Proceedings of the 5th International Conference of Software Engineering: IEEE, 1981, pp. 215-223.

[179] J. Yang, C. A. Shaffer, and L. S. Heath, "A Data Structure Visualisation System," in Graph Drawing, vol. 1027, Lecture Notes in Computer Science, F. J. Brandenburg and J. Franz, Eds. Passau: Springer-Verlag, 1995, pp. 520-523.

[180] P. Young, "HELP! - GraphTool for Java," Research Institute for Software Evolution, Durham, Manual September 1999.

[181] E. Yourdon, Model Structured Analysis. Englewood Cliffs: Prentice Hall, 1989.